OSEKturbo OS/MPC5xx v.2.2.1

Technical Reference

Because of last-minute software changes, some information in this manual may be inaccurate. Please read the readme.txt file for the latest information.

Revised: June 2003

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

Legal Notices

Important Notice to Users

Trademarks

For More Information: www.freescale.com

Contents

Freescale Semiconductor, Inc.

For More Information: www.freescale.com

Freescale Semiconductor, Inc.

For More Information: www.freescale.com

1

Introduction

This Technical Reference describes the OSEKturbo OS/MPC5xx, a version of the OSEK[1] Operating System (OSEK OS) for PowerPC providing high speed performance and low RAM usage. All the system mechanisms, particularities, services and programming techniques are described in detail with numerous examples. Data for performance characteristics and memory requirements are provided.

"Operating System Architecture" chapter gives a high level description of the OS architecture and presents OS Conformance Classes.

"Task Management" chapter explains the task concept in OSEK and all other questions related to tasks.

"Scheduler" chapter provides a description of scheduling policies in OSEK OS.

"Interrupt Processing" chapter highlights the OSEK approach to interrupt handling.

"Resource Management" chapter describes resource management and task coordination by resources.

"Counters and Alarms" chapter describes usage of these control mechanisms in OSEK OS.

"Events" chapter is devoted to event management and task coordination by events.

[1.] The term OSEK means 'Open systems and the corresponding interfaces for automotive electronics' (in German). A real-time operating system, software interfaces and functions for communication and network management tasks are thus jointly specified within the OSEK standard.

"Communication" chapter describes a message concept in OSEK and its usage.

"Error Handling and Special Routines" chapter describes support provided for the user to debug an application and handle errors.

"System Configuration" chapter describes possible OSEK OS versions, configuration options and the configuration mechanism.

"System Objects Definition" chapter describes the objects controlled by the Operating System: tasks, resources, alarms, messages, counters, ISRs and even the OS itself are considered as system objects.

"Building of Application" chapter contains information on how to build a user's application using the OSEK OS. It also describes memory requirements.

"PowerPC Platform-Specific Features" chapter discusses special OSEK OS features for different MCU types and issues connected with porting applications to these MCUs.

"Application Troubleshooting" chapter contains useful information for debugging applications developed using the OSEK OS.

"System Services" chapter provides a detailed description for all OSEK Operating System run-time services, with appropriate examples.

"Debugging Application" chapter provides information about preparation of all data required for the OSEK aware debugger to display information about an application in OSEK terms.

"Sample Application" appendix contains the text and listing of a sample customer application developed using the OSEK OS.

"System Service Timing" appendix provides information about OS services execution time.

"Memory Requirements" appendix provides information about the amount of ROM and RAM directly used by various versions of the OSEK OS.

"System Generation Error Messages" appendix explains OSEK OS System Generator error messages.

The "Introduction" chapter consists of the following sections:

- OSEK OS Overview
- Typographical Conventions
- References
- Definitions, Acronyms and Abbreviations
- Technical Support Information

OSEK OS Overview

The OSEK Operating System is a real-time operating system which conforms to the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification.

The OSEK OS meets the following requirements:

- The OS is fully configured and statically scaled;
- The OS performance parameters are well known;
- The most part of the OS is written in strict correspondence with ANSI C standard, the OS and the application on its basis can be easily ported from one platform to another.

A wide range of scalability, a set of system services, various scheduling mechanisms, and convenient configuration features make the OSEK Operating System feasible for a broad spectrum of applications and hardware platforms.

The OSEK OS provides a pool of different services and processing mechanisms for task management and synchronization, data exchange, resource management, and interrupt handling. The following features are provided for the user:

Task Management

- Activation and termination of tasks;
- Management of task states, task switch.

Scheduling Policies

- Full-, non-, and mixed-preemptive scheduling techniques.

Event Control

- Event Control for task synchronization.

Interrupt Management

- Services for disabling/enabling all interrupts;

- Services for disabling/enabling interrupts of category 2;

Resource Management

- Control of mutually exclusive access to jointly used resources or devices, or for control of a program flow.

Communication[1]

- Data exchange between tasks and/or ISRs;

Counter[2] and Alarm Management

- The counter management provides services for execution of recurring events;

- The alarm management is based on the counter management. The alarm management allows the user to perform link task activation or event setting to a certain counter value. These alarms can be defined as either single (one-shoot) or cyclic alarms. Expiration of a preset relative counter value, or the fact that a preset absolute counter value is reached, results in activation of a task, or setting a task event.

- TimeScale[3] enables periodic activations of tasks in accordance with a static defined schedule.

Error Treatment

- Mechanisms supporting the user in case of various errors.

ORTI Subsystem

- The ORTI provides an interface to Operating System run-time data for "OSEK aware" debuggers.

---

[1] The Communication part of the OSEK Operating System conforms to the *OSEK/VDX Communication, v.2.2.2, 18 December 2000* specification.

[2] The Counter Management part of the OSEK Operating System conforms to the *OSEK Operating System, Application Program Interface, v.1.00, 11 September 1995* specification.

[3] TimeScale is an OSEKturbo extension of OSEK OS.

The OSEK Operating System is scaled in two ways: either by changing the set of system services or through the so-called Conformance Classes. They are available to meet different requirements concerning the OS functionality and capability. These Conformance Classes differ not only in the number of services they provide, but also in their capabilities and scalability. The classes are based on one another in upwardly compatible fashion. (see "Conformance Classes")

The OSEK OS is built according to the user's configuration instructions while the system is generated. Both system and application parameters are configured statically. Therefore, a special tool called the System Generator is used for this purpose. Special statements are designed to tune any parameter. The user must only edit the definition file, run the System Generator and then assemble the resulting files and the application ones. Thus, the user can adapt the Operating System for the control task and the target hardware. The OS cannot be modified later at the run time.

Typographical Conventions

This Technical Reference employs the following typographical conventions:

Boldface type

Bold is used for important terms, notes and warnings.

Italics

Italics are used for all OSEK names of directives, macros, constants, routines and variables.

`Courier` font

The courier typeface is used for code examples in the text.

References

[1]   OSEK/VDX Operating System, v.2.2, 10 September 2001

[2]   OSEK/VDX System Generation OIL: OSEK Implementation Language, v.2.3, 10 September 2001

[3]   OSEK/VDX Communication, v.2.2.2, 18 December 2000

[4]   OSEK Operating System, Application Program Interface, v.1.00, 11 September 1995

[5]   ORTI: OSEK Run Time Interface, v.2.0 (Draft c), 23 June 1999

[6]   OSEK/VDX OSEK Run Time Interface (ORTI), Part A: Language Specification, v. 2.1.1, 4 March 2002

[7]   OSEK/VDX OSEK Run Time Interface (ORTI), Part B: OSEK Objects and Attributes, v. 2.1, 17 April 2002

[8]   OSEKturbo OS/MPC5xx v.2.2.1, User's Manual

Definitions, Acronyms and Abbreviations

The following acronyms and abbreviation are used in this Technical Reference.

API	Application Program Interface (a set of data types and functions)
BCC	Basic Conformance Class, a defined set of functionality in OSEK, for which waiting state of tasks is not permitted
BT	Basic task (the task which never has waiting state)
CCCB	OSEK Conformance Communication Class B
CPU	Central Processor Unit
ECC	Extended Conformance Class, a defined set of functionality in OSEK, for which waiting state of tasks is permitted
ET	Extended Task (the task which may have waiting state)
HW	Hardware
ID	Identifier, an abstract identifier of a system object
ISR	Interrupt Service Routine
MGT5x00	MGT5100 or MGT5200 microcontroller
MCU	Microcontroller Unit
N/A	Not applicable
OIL	OSEK Implementation Language
ORTI	OSEK Run Time Interface

OS	Operating System
OSEK	Open systems and the corresponding interfaces for automotive electronics (in German)
RAM	Random Access Memory
ROM	Read Only Memory
SG, SysGen	System Generator Utility
SW	Software

Technical Support Information

To order Motorola/Metrowerks products or literature, consult your local sales representative.

For technical support please use:

US
Tel: +1 512 997 4700
Fax: +1 512 997 4901
Email: support@metrowerks.com

Europe
Tel: +41 61 69 07 505
Fax: +41 61 69 07 501
Email: support_europe@metrowerks.com

Web: http://www.metrowerks.com/MW/Support

Download updates at
http://www.metrowerks.com/MW/Support/Download

2

Operating System Architecture

This chapter gives a high level description of the OS architecture and presents the OS Conformance Classes.

This chapter consists of the following sections:

- Processing Levels
- Conformance Classes
- OSEK OS Overall Architecture
- Application Program Interface

Processing Levels

The OSEK Operating System provides a pool of different services and processing mechanisms. It serves as a basis for application programs which are independent of each other, and provides their environment on a processor. The OSEK OS enables controlled real-time execution of several processes which virtually run in parallel.

The OSEK Operating System provides a defined set of interfaces for the user. These interfaces are used by entities competing for the CPU. There are two types of entities:

- Interrupts (service routines managed by the Operating System);
- Tasks (basic tasks and extended tasks).

The highest processing priority is assigned to the interrupt level, where interrupt service routines (ISR) are executed. The interrupt services may call a number of operating system services. The processing level of the operating system has the priority immediately below the former one. This is the level on which the operating system works: task management procedures, scheduler and system services. Immediately below there is the task level on

---

which the application software is executed. Tasks are executed according to their user assigned priority. A distinction is made between the management of tasks with and without *waiting* state (*Extended* and *Basic Tasks*, see "Task Concept").

The following set of priority rules has been established:

- interrupts have precedence over tasks;
- the interrupt priority is defined by specific hardware conditions;
- for the items handled by the OS, bigger numbers refer to higher priorities;
- the task's priority is statically assigned by the user.

The Operating System provides services and ensures compliance with the set of priority rules mentioned above.

Conformance Classes

Various requirements of the application software for the system, and various capabilities of a specific system (e.g. processor type, amount of memory) demand different features of the operating system. These operating system features are described as *Conformance Classes* (CC). They differ in the number of services provided, their capabilities and different types of tasks.

The Conformance classes were created to support the following objectives:

- providing convenient groups of operating system features for easier understanding and discussion of the OSEK operating system.
- allowing partial implementations along pre-defined lines. These partial implementations may be certified as OSEK compliant.
- creating an upgrade path from the classes of less functionality to the classes of higher functionality with no changes to the application using OSEK related features.

The required Conformance Class is selected by the user at the system generation time and cannot be changed during execution.

Definition of the functionalities provided by each Conformance Class depends on the properties of the tasks and the scheduling

behavior. As the task properties (Basic or Extended, see "Task Concept") have a distinct influence on CC, they also assume part of their names. There are Basic-CC and Extended-CC, and each of these groups can have various "derivatives".

The Conformance classes are determined by the following attributes:

- Multiply requesting of task activation- not supported by the OSEKturbo;

- Task types (see "Task Concept");

- Number of tasks per priority.

Figure 2.1   Restricted Upward Compatibility for Conformance Classes
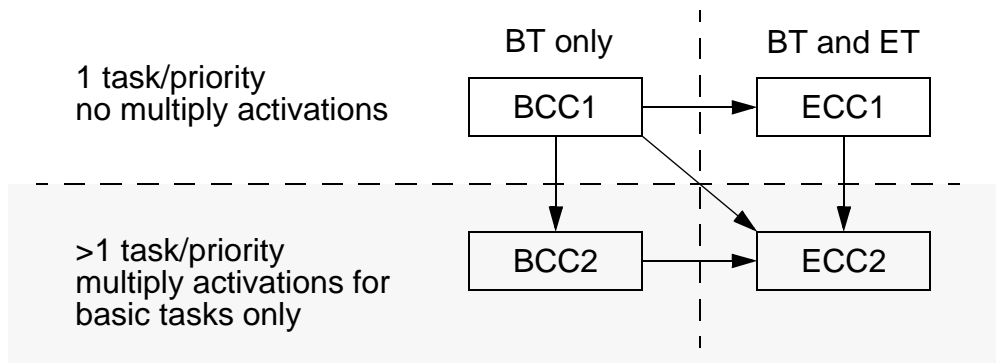
The OSEK OS specification defines the following Conformance Classes: BCC1, BCC2, ECC1, ECC2.The OSEKturbo does not support multiply activation and therefore it doesn't have BCC2 and ECC2 classes.

The OSEKturbo OS supports the following Conformance Classes:

- BCC1 – only Basic tasks, limited to one activation request per task and one task per priority, and all tasks have different priorities;

- ECC1 – like BCC1, plus Extended tasks.

For More Information: www.freescale.com

Table 2.1 indicates the minimum resources to which an application may resort, determined for each Conformance Class in the OSEK OS.

Table 2.1   OSEK OS Conformance Classes

	BCC1	BCC2	ECC1	ECC2
Multiple activation of tasks	no	yes	BT: no, ET: no	BT: yes, ET: no
Number of tasks which are not in *suspended* state	>=8		>= 16, any combination of BT/ET	
Number of tasks per priority	1	>1	1 (both BT/ ET)	>1 (both BT/ ET)
Number of events per task	-		BT: no ET: >= 8	
Number of task priorities	>=8		>=16	
Resources	only Scheduler	>= 8 resources (including Scheduler)		
Internal Resources	>=2			
Alarm	>= 1 single or cyclic alarm			
Messages	possible			

The system configuration option *CC* (specified by the user) defines the class of the overall system. In the OSEKturbo OS this option can have the values *BCC1* and *ECC1* or it can be set to *AUTO* (see "OS Definition").

Maximal numbers of the OSEKturbo OS/MPC5xx system objects are indicated in Table 2.2.

Table 2.2   OSEKturbo OS/MPC5xx Maximal System Resources

Number of task's priorities	64
Number of tasks which are not in suspended state	64
Number of events per task	32
Number of resources (including RES_SCHEDULER)	255
Number of Apllication Modes	8
Number of other OSEK objects (alarms, messages, counters)	254

OSEK OS Overall Architecture

The OSEK OS is a real-time operating system which is executed within a single electronic control unit. It provides local services for the user's tasks. The OSEK OS consists of the following components:

- *Scheduler* controls the allocation of the CPU for different tasks;
- *Task management* provides operations with tasks;
- *ISR management* provides entry/exit frames for interrupt service routines and supports CPU interrupt level manipulation;
- *Resource management* supports a special kind of semaphore for mutually exclusive access to shared resources;
- *Local communication* provides message exchange between tasks;
- *Counter management* provides operations on objects like timers and incremental counters;
- *Alarm management* links the tasks and counters;
- *Error handlers* handle the user's application errors and internal errors, and provide recovery from the error conditions;
- *Hook routines* provide additional debugging features
- *System start-up* initializes the data and starts the execution of the applications;
- *System timer* provides implementation-independent time management.

As you can see in Table 2.1, the Conformance Classes, in general, differ in the degree of services provided for the task management and scheduling (the number of tasks per priority, multiple requesting, Basic/Extended Tasks). In higher CC an advanced functionality is added for the resource management and event management only. But even in BCC1 the user is provided with almost all the OSEK OS service mechanisms.

The OSEK Operating System is not scaled through the Conformance Classes only, but it also has various extensions which can be in any Conformance Class. These extensions affect memory requirements and overall system performance. The extensions can be turned on or off with the help of the corresponding system configuration options. They all are described in "System Objects Definition".

For More Information: www.freescale.com

Since the OSEK Operating System is fully statically configured, the configuration process is supported by the *System Generator* (SG). This is a command-line utility, which processes system generation statements defined by the user in a special file. These statements fully describe the required system features and application object's parameters. The SG produces a header file (osprop.h) which is used for system compilation and C-code files to be compiled together with the other user's source code. The produced code consists of C-language definitions and declarations of data as well as C-preprocessor directives. See "System Configuration" and "Building of Application" for details about system generation.

Application Program Interface

The OSEK Operating System establishes the Application Program Interface (API) which must be used for all the user actions connected with system calls and system objects. This API defines the data types used by the system, the syntax of all run-time service calls, declarations and definitions of the system.

The OSEK OS data types are described in subsections dedicated to the corresponding mechanisms. The syntax of system calls and system configuration statements is described briefly in the corresponding subsections and in detail in "System Objects Definition" and "System Services".

NOTE   The user's source code shall strictly correspond to the rules stated in this Technical Reference.

The OSEK OS may be compiled in *Extended Status*. It means that an additional check is made within all OS activities and extended return codes are returned by all the OS services to indicate errors if any. See "System Services" and "Error Handling" about Extended Status return values. In order to provide the Extended Status in the system, the configuration option *STATUS* must be set to *EXTENDED* at the configuration stage.

The OSEKturbo OS provides support for the "OSEK aware" debuggers by means of the OSEK Run Time Interface (ORTI). See

For More Information: www.freescale.com

"Debugging Application" and "Global System Attributes" for details.

For More Information: www.freescale.com

3

Task Management

This chapter describes the task concept of the OSEK and all other questions related to tasks.

This chapter consists of the following sections:

- Task Concept
- Task Priorities
- Tasks Stacks
- Programming Issues

Task Concept

Complex control software can be conveniently subdivided into parts executed according to their real-time requirements. These parts can be implemented by means of tasks. The task provides the framework for execution of functions. The Operating System provides parallel and asynchronous task execution organization by the scheduler.

Two different task concepts are provided by the OSEK OS:

- Basic Tasks (BT);
- Extended Tasks (ET).

The Basic Tasks release the processor only if:

- they are being terminated,
- the OSEK OS is executing higher-priority tasks, or
- interrupt have occurred.

The Extended Tasks differ from the Basic Tasks by being allowed using additional operating system services which may result in *waiting* state. *Waiting* state allows the processor to be freed and reassigned to a lower-priority task without the necessity to terminate the Extended Task.

The task type is determined automatically. If a TASK object has a reference to EVENT, the task is considered to be Extended.

Both types of tasks have their advantages which must be compared in context of application requirements. They both are justified and supported by the OSEK operating system.

Every task has a set of related data: task description data located in ROM and task state variables in RAM. Also, every extended task has its own stack assigned.

Every running task is represented by its run-time context. This refers to CPU registers and some compiler-dependent 'pseudoregisters' in RAM. When the task is interrupted or preempted by another task, the run-time context is saved.

The task can be in several states since the processor can execute only one instruction of the task at any time, but at the same time several tasks may compete for the processor. The OSEK OS is responsible for saving and restoring the task context in conjunction with state transitions whenever necessary.

Extended Tasks

The Extended Tasks have four task states:

- *running*
  In running state the CPU is assigned to the task so that its instructions can be executed. Only one task can be in this state at the same point in time, while all the other states can be adopted simultaneously by several tasks.

- *ready*
  All functional prerequisites for transition into running state are met, and the task only waits for allocation of the processor. The scheduler decides which of ready tasks is executed next.

- *waiting*
  A task cannot be executed (any longer), because it has to wait for at least one event (see "Events").

- *suspended*
  In suspended state the task is passive and does not occupy any resources, merely ROM.

Figure 3.1   Status Model with Task Transitions for an Extended Task
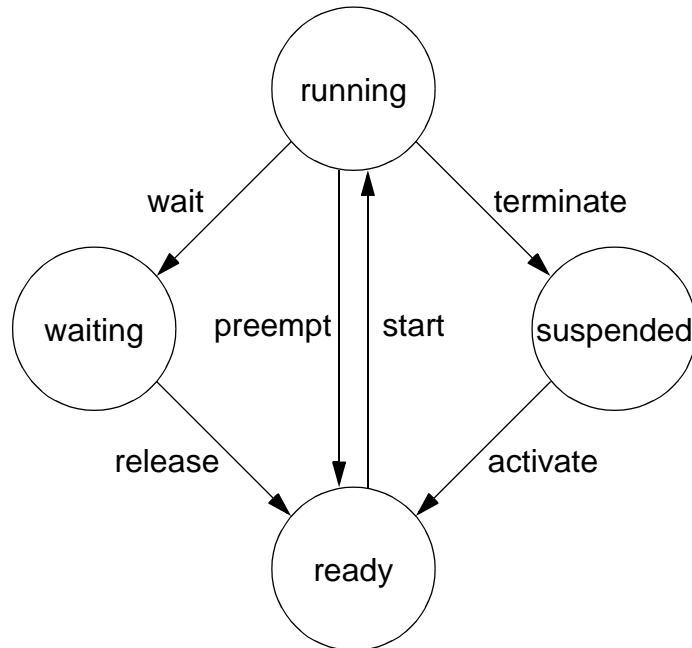
Table 3.1   States and Status Transitions for an Extended Task

Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the *ready* list by a system service.
start	ready	running	A *ready* task selected by the scheduler is executed.
wait	running	waiting	To be able to continue an operation, the *running* task requires an event. It causes its transition into *waiting* state by using a system service.
release	waiting	ready	Events have occurred which a task has been waiting for.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into *ready* state.
terminate	running	suspended	The *running* task causes its transition into *suspended* state by a system service.

Termination of tasks is only possible if the task terminates itself ('self-termination').

There is no provision for a direct transition from *suspended* into *waiting* state. This transition is redundant and would make the scheduler more complicated. *Waiting* state is not directly entered from *suspended* state since the task starts and explicitly enters *waiting* state on its own.

Basic Tasks

The state model for the Basic Tasks is nearly identical to the one for the Extended Tasks. The only exception is absence of *waiting* state.

- *running*
  In running state the CPU is assigned to the task so that its instructions can be executed. Only one task can be in this state at the same point in time, while all the other states can be adopted simultaneously by several tasks.

- *ready*
  All functional prerequisites for transition into running state are met, and the task only waits for allocation of the processor. The scheduler decides which of ready tasks is executed next.

- *suspended*
  In suspended state the task is passive and does not occupy any resources, merely ROM.

Table 3.2   States and Status Transitions for a Basic Task

Transition	Former state	New state	Description
activate	suspended	ready	A new task is entered into the *ready* list by a system service.
start	ready	running	A *ready* task selected by the scheduler is executed.
preempt	running	ready	The scheduler decides to start another task. The *running* task is put into *ready* state.
terminate	running	suspended	The *running* task causes its transition into *suspended* state by a system service.

Figure 3.2   Status Model with Task Transitions for a Basic Task
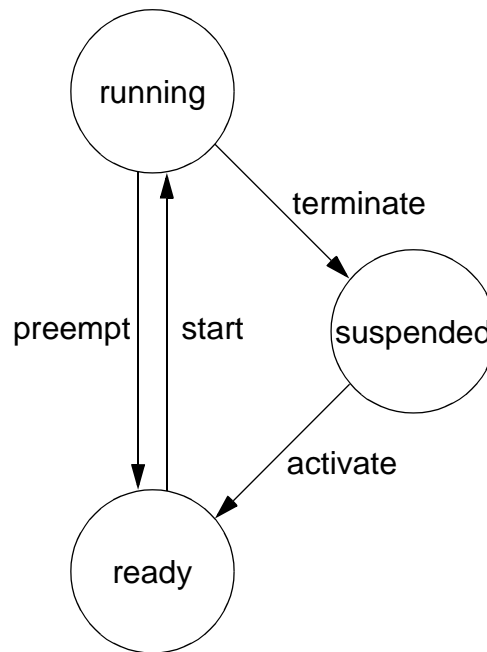
Task Priorities

The OSEK OS specifies the value 0 as the lowest task priority in the operating system. Accordingly bigger numbers define higher task priorities.

The task priorities defined in OIL are not intersected with ISR priorities. Interrupts have a separate priority scale. All task priorities are lower than any ISR priorities and the scheduler priority.

In the OSEK OS the priority is statically assigned to each task and it cannot be changed at run-time. A dynamic priority management is not supported. However, in particular cases the operating system can change task priority. In this context, please refer to "Priority Ceiling Protocol".

When rescheduling is performed, the scheduler always switches to the task with the highest priority among the *ready* tasks and the *running* one.

Tasks Stacks

Stack Allocation

Each extended task has its own statically allocated stack.

The minimal size of the task stack depends on:

- the scheduling policy (non-preemptable or preemptable task);
- the services used by the task;
- the interrupt and error handling policy;
- the processor type.

The recommended values of the minimal task stack size are provided in "Stack Size".

NOTE   If the task stack is less than the required value for the given application, it may cause unpredictable behavior of the task and a system crash.

Single Stack

The single stack is used for the basic tasks in all configurations. In the BCC1 conformance class there is only the single stack in the OS. It is used for all tasks, interrupts and dispatcher.

In the ECC1 class all the basic tasks use the single stack. But the extended tasks have their own stacks. Interrupts have a separate stack as well.

The OSEKturbo OS uses the main application stack for a single stack. Its size is defined by the user at the link stage.

For More Information: www.freescale.com

Programming Issues

Configuration Options

The following system configuration options affect the task management:

- *STATUS*
  Specifies error checking at run-time.

- *StackOverflowCheck*
  Turns on stack overflow runtime checking and stack usage services.

- *CC*
  Specifies the conformance class. If AUTO, the conformance class is defined according to the task definitions.

Data Types

The OSEK OS establishes the following data types for the task management:

- *TaskType*
  The abstract data type for task identification.

- *TaskRefType*
  The data type to refer the variables of the *TaskType* data type. Reference to the *TaskType* variable can be used instead of the *TaskRefType* variable.

- *TaskStateType*
  The data type for the variables for storage of the task state;

- *TaskStateRefType*
  The data type to refer the variables of the *TaskStateType* data type. Reference to the *TaskStateType* variable can be used instead of the *TaskStateRefType* variable.

Only these data types may be used for operations with tasks.

Task Definition

Every task in an application is generated using the *TASK* system generation object with the set of properties in OIL file. These properties define the task behavior and the resource allocation

method. Each task property has its own name, and the user defines the task's features by setting the corresponding properties in the task definition. See also "System Objects Definition".

The task definition looks like the following:

```
TASK TASKSENDER {
  PRIORITY = 5;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  RESOURCE = MYRESOURCE;
  RESOURCE = SECONDRESOURCE;
  EVENT = MYEVENT;
  STACKSIZE = 256;
  ACCESSOR = SENT {
    MESSAGE = MYMESSAGE;
    WITHOUTCOPY = TRUE;
    ACCESSNAME = "MessageBuffer";
  };
};
```

A description of possible task properties is indicated in Table 3.3.

Table 3.3   Task Properties

Object Parameters	Possible Values	Description
Standard Attributes		
PRIORITY	integer [0...0x7FFFFFFF]	Defines the task priority. The lowest priority has the value 0
SCHEDULE	FULL, NON	Defines the run-time behavior of the task
AUTOSTART	TRUE, FALSE	Defines whether the task is activated during the system start-up procedure or not
APPMODE	name of APPMODE	Defines an application mode in which the task is auto-started
ACTIVATION	1	Specifies the maximum number of queued activation requests for the task (the OSEKturbo does not allow multiply activations)

Table 3.3   Task Properties

RESOURCE	name of RESOURCE	Resources accessed by the task. There can be several resource references
EVENT	name of EVENT	Event owned by the task. There can be several event references
ACCESSOR	SENT, RECEIVED	Defines the type of usage of the message
MESSAGE	name of MESSAGE	Specifies the message to be sent or received by the task
WITHOUTCOPY	TRUE, FALSE	Defines whether a local copy of the message is used or not
ACCESSNAME	string	Defines the reference which can be used by the application to access the message data
OSEKturbo Specific Attribute		
STACKSIZE	integer	Defines the size of the task stack in bytes (only for the Extended Tasks)

The application definition file contains one such statement per task. The task generation statement is described in detail in "System Objects Definition".

The constructional statement *DeclareTask* may be used for compatibility with the previous OSEK versions. It may be omitted in the application code.

DeclareTask is as follows:

```
DeclareTask( <TaskName> );
```

Run-time Services

The OSEK OS provides a set of services for the user to manage tasks. A detailed description of these services is provided in "System Services". Below is only a brief list of them.

Table 3.4   Task Management Run-time Services

Service Name	Description
ActivateTask	Activates the task, i.e. moves it from *suspended* to *ready* state
TerminateTask	Terminates the running task, i.e. moves it from *ready* to *suspended* state

Table 3.4   Task Management Run-time Services

Service Name	Description
ChainTask	Terminates the running task and activates the new one immediately
Schedule	Yields control to a higher-priority ready task (if any)
GetTaskId	Gets the identifier of the running task
GetTaskState	Gets the status of a specified task

Examples of using the run-time services are provided in "Examples for Task Management Services".

Constants

The following constants are used within the OSEK Operating System to indicate the task states:

- *RUNNING*
  The constant of data type *TaskStateType* for task state *running*
- *WAITING*
  The constant of data type *TaskStateType* for task state *waiting*
- *READY*
  The constant of data type *TaskStateType* for task state *ready*
- *SUSPENDED*
  The constant of data type *TaskStateType* for task state *suspended*

These constants can be used for the variables of *TaskStateType*.

The following constant is used within the OSEK OS to indicate the task:

- *INVALID_TASK*
  The constant of data type *Task Type* for an undefined task

Conventions

Within the OSEK OS application a task should be defined according to the following pattern:

```
TASK ( TaskName )
{
```

For More Information: www.freescale.com

```
...
}
```

The name of the task function will be generated from *TaskName* by macro *TASK*.

4

Scheduler

This chapter provides a description of scheduling policies in OSEK OS.

This chapter consists of the following sections:

- General
- Scheduling Policy
- Programming Issues

General

The algorithm deciding which task has to be started and triggering all necessary OSEK Operating System internal activities is called *scheduler*. It performs all actions to switch the CPU from one instruction thread to another. It is either switching from task to task or from ISR back to a task. The task execution sequence is controlled on the base of task priorities (see section "Task Priorities") and the scheduling policy used.

The scheduler is activated whenever a task switch is possible according to the scheduling policy. The principle of multitasking allows the operating system to execute various tasks concurrently. The sequence of their execution depends on the scheduling policy, therefore it has to be clearly defined.

Scheduler also provides the endless idle loop if there is no task ready to be running. It may occur, when all tasks are in the *suspended* or *waiting* state until the awakening signal from an Interrupt Service Routine occurs. In this case there is no currently running task in the system, and the scheduler occupies the processor performing an endless loop until the ISR awakes a task to be executed. It is possible to call a special user's hook from the scheduler idle loop. This property is turned on via the system configuration option *IdleLoopHook*. An instruction that puts the CPU in low power mode may be inserted into idle loop to reduce power

consumption. This property is turned on via the system configuration option *HCLowPower*.

The scheduler can be treated as a specific resource that can be occupied by any task. See "Scheduler as a Resource" for details.

The scheduling policy and some scheduler-related parameters are defined by the user, see "Global System Attributes".

Scheduling Policy

The scheduling policy being used determines whether execution of a task may be interrupted by other tasks or not. In this context, a distinction is made between full-, non- and mixed-preemptive scheduling policies. The scheduling policy affects the system performance and memory resources. In the OSEK Operating System, all listed scheduling policies are supported. Each task in an application may be preemptable or not. It is defined via the appropriate task property (preemptable/non-preemptable).

Note that the interruptability of the system depends neither on the Conformance Class, nor on the scheduling policy.

The desired scheduling policy is defined by the user via the tasks configuration option *SCHEDULE*. The valid values are – *NON and FULL.* If all tasks use NON scheduling, scheduler works as non-preemptive. If all tasks use FULL scheduling, scheduler works as full-preemptive. If some tasks use NON and other tasks use FULL scheduling, scheduler works as mixed-preemptive.

Non-preemptive Scheduling

The scheduling policy is considered as non-preemptive, if a task switch is only performed via one of a selection of explicitly defined system services (explicit point of rescheduling).

Non-preemptive scheduling imposes particular constraints on the possible timing requirements of tasks. Specifically, the lower priority non-preemptable section of a running task delays the start of a task with higher priority, up to the next point of rescheduling. The time diagram of the task execution sequence for this policy looks like the following:

For More Information: www.freescale.com

Figure 4.1   Non-preemptive Scheduling
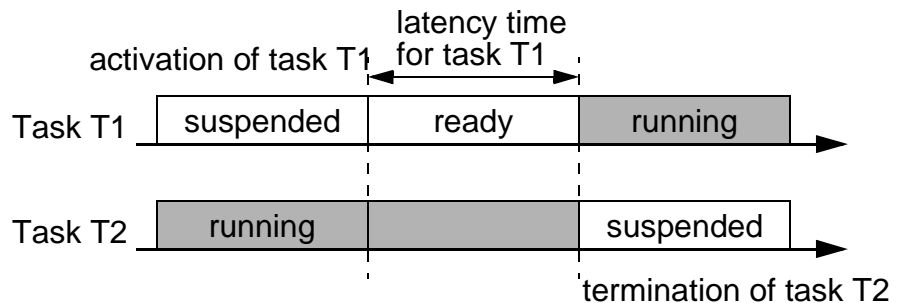
Task T2 has lower priority than task T1. Therefore, it delays task T1 up the point of rescheduling (in this case termination of task T2).

Only four following *points of rescheduling* exist in the OSEK OS for non-preemptive scheduling:

- Successful termination of a task (via the *TerminateTask* system service);
- Successful termination of a task with explicit activation of a successor task (via the *ChainTask* system service);
- Explicit call of the scheduler (via the *Schedule* system service);
- Explicit wait call, if a transition into the waiting state takes place (via the *WaitEvent* system service, Extended Tasks only).

In the non-preemptive system, all tasks are non-preemptable and the task switching will take place exactly in the listed cases.

Full-preemptive Scheduling

Full-preemptive scheduling means that a task which is presently *running* may be rescheduled at any instruction by the occurrence of trigger conditions preset by the operating system. Full-preemptive scheduling will put the *running* task into the *ready* state as soon as a higher-priority task has got *ready*. The task context is saved so that the preempted task can be continued at the location where it was interrupted.

With full-preemptive scheduling, the latency time is independent of the run time of lower priority tasks. Certain restrictions are related to the enhanced complexity of features necessary for synchronization between tasks. As each task can theoretically be rescheduled at any location, access to data that are used jointly with other tasks must be synchronized.

In a full-preemptive system all tasks are preemptable.

Figure 4.2   Full-preemptive Scheduling
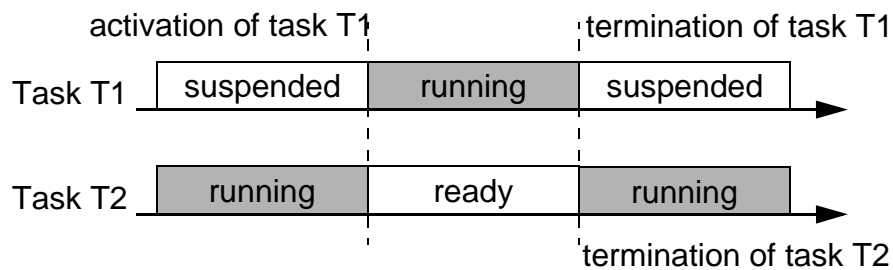
Mixed-preemptive Scheduling

If full-preemptive and non-preemptive scheduling principles are to be used for execution of different tasks on the same system, the resulting policy is called "mixed-preemptive" scheduling. The distinction is made via the task property (preemptable/non-preemptable) of the running task.

The definition of a non-preemptable task makes sense in a full-preemptive operating system in the following cases:

• if the execution time of the task is in the same magnitude of the time of a task switch,

• if the task must not be preempted.

Many applications comprise only a few parallel tasks with a long execution time, for which a full-preemptive scheduling policy would be convenient, and many short tasks with a defined execution time where non-preemptive scheduling would be more

efficient. For this configuration, the mixed-preemptive scheduling policy was developed as a compromise.

Groups of Tasks

The operating system allows tasks to combine aspects of preemptive and non-preemptive scheduling by defining groups of tasks. For tasks which have the same or lower priority as the highest priority within a group, the tasks within the group behave like non-preemptable tasks: rescheduling   will only take place at the points of rescheduling described in "Non-preemptive Scheduling" For tasks with a higher priority than the highest priority within the group, tasks within the group behave like preemptable tasks (see "Full-preemptive Scheduling").

Chapter "Internal resources" describes the mechanism of defining groups by the instrumentality of internal resources.

Programming Issues

Configuration Options

The following system configuration options are intended to define scheduler properties:

- *CC*
  Specifies conformance class. If AUTO, conformance class is defined according to tasks definitions.

- *IdleLoopHook*
  If this option is turned on, then user supplied hook will be called from the scheduler idle loop.

- *HCLowPower*
  If this option is turned on, an instruction that puts the CPU in low power mode is used instead of the scheduler's idle loop.

- *ResourceScheduler*
  If this option is set to FALSE then *RES_SCHEDULER* is not supported by OS.

Run-time Services

The scheduler is not accessed by the user directly. The user can only pass the CPU control to the scheduler by means of the *Schedule* system service. This leads to task rescheduling if there is a ready task of higher priority.

The scheduler can be used by the programmer as a resource. To provide this possibility, the services *GetResource* and *ReleaseResource* with the constant *RES_SCHEDULER* as a parameter can be called by a task. It means that the task cannot be preempted by any other task after the scheduler occupation, before the corresponding call *ReleaseResource* is performed. While the task occupies the scheduler, it has the highest priority and, therefore, cannot be preempted by other tasks (only ISRs can get the CPU control during this period). Such programming practice can be used for important critical sections of code.

See the example:

```
GetResource( RES_SCHEDULER );
...
/* Critical section */
/* this code cannot be interrupted by any other task */
...
ReleaseResource( RES_SCHEDULER );
```

5

Interrupt Processing

This chapter highlights the OSEK approach to interrupt handling.

This chapter consists of the following sections:

- General
- ISR Categories
- Interrupt Level Manipulation
- ISR Stack
- Interrupt Dispatcher

General

Interrupt processing is an important part of any real-time operating system. An *Interrupt Service Routine (ISR)* is a routine which is invoked from an interrupt source, such as a timer or an external hardware event. ISRs have higher priority than all tasks and the scheduler. Addresses of ISRs should be pointed to in the vector table.

In OSEKturbo OS/MPC5xx in ECC1 class all ISRs of category 2 use the separate stack (*ISR stack*) which is used only by ISRs during their execution. The size of the ISR stack is defined by the user.

OSEKturbo OS/MPC5xx supports nested interrupts up to 17 levels (up to 49 levels if *Enhanced InterruptDispatcher* is used). Since the OS provides the means to switch the stack and to control the interrupt mask, such nested interrupts, if written correctly, could be treated as a single one. To not waste a task stack space for nested interrupts or complicated ISR, the ISR stack is used.

According to the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification there are no services for manipulation of CPU and/or OS interrupt levels directly. Therefore nested interrupts with the same hardware levels can not occur. For CPU with one hardware level nested interrupts are forbidden. For multilevel

---

modes nested interrupts with different priority are allowed. Application is not allowed to manipulate interrupt enabling bits in the CPU state registers.

NOTE   If application manipulates directly with CPU registers that control interrupts then OS behavior is unpredictable. There is an exception for this rule, please see Enhanced Interrupt Controller

---

ISRs can communicate with tasks in the OSEK OS by the following means:

- ISR can activate a task;
- ISR can send/receive an unqueued messages in WithCopy configuration;
- ISR can trigger a counter;
- ISR can get Task ID;
- ISR can get state of the task;
- ISR can set event for a task;
- ISR can get event mask of the task;
- ISR can manipulate alarms;

Interrupts cannot use any OS services except those which are specially allowed to be used within ISRs. When using other services, the system behavior will be unpredictable. In the Extended (debugging) status of the Operating System, the error will be reported in such a case. See Table 5.1 on page 52 and "System Services" for details.

ISR Categories

In the OSEK Operating System two types of Interrupt Service Routines are considered.

ISR Category 1

ISRs of these type are executed on the current stack. In this case, if the ISR uses the stack space for its execution, the user is responsible for the appropriate stack size. Only 6 Interrupt Management

services (enabling/disabling interrupts) are allowed in ISRs of category 1 (see Table 5.1). After the ISR is finished, processing continues exactly at the instruction where the interrupt occurred, i.e. the interrupt has no influence on task management.

The following statements are used to define ISR category 1.

```
ISR( ISR_handler )
{
...
/* the code without any OS service calls */
...
}
```

WARNING!   ISR category 1 should have the priority higher than any ISR of category 2. If ISR category 1 is interrupted by ISR category 2, rescheduling may take place at the end of ISR category 2 execution and ISR category 1 execution will be suspended therefore.

ISR Category 2

In ISR category 2 the OSEK Operating System provides an automatic switch to the ISR stack (for ECC1 only) and enters OSEK OS execution context. After that, any user's routine can be executed, including allowed OS calls (to activate a task, send a message or trigger a counter). See "Run-time Services" for the list of services allowed for ISR. At the end of the ISR, the System automatically switches back to the task stack and restores context.

The following statements are used to define ISR category 2.

```
ISR( ISR_handler )
{
  /* the code with allowed OS calls */
}
```

Inside the ISR, no rescheduling will take place. Rescheduling may only take place on termination of the ISR of category 2 if a preemptable task has been interrupted.

WARNING! Interrupts which can not be disabled, such as NMI or any synchronous exception, shall not be assigned to ISRs of category 2. If OS could not disable ISR category 2, then OS will not be able to protect it's critical code sections and will crash.

Interrupt Level Manipulation

Direct manipulation with CPU interrupt flags or levels is strictly forbidden. The user can not define values of the interrupt masks directly. Interrupts are enabled during task execution. Interrupts can be disabled via disable/enable interrupt API functions or by using resource mechanism.

DisableAllInterrupts service can be used to temporary disable all interrupts. To return to previous interrupt status *EnableAllInterrupts* service must be called after it in frame of task or ISR where *DisableAllInterrupts* is called.

SuspendAllInterrupts and *ResumeAllInterrupts* pair has the same effect as *DisableAllInterrupts - EnableAllInterrupts* pair but allows nesting of pairs.

SuspendOSInterrupts service can be used to temporary disable all interrupts of category 2. To return to previous interrupt status *ResumeOSInterrupts* service must be called after it in frame of task or ISR where *SuspendOSInterrupts* is called.

Resources can be used to temporary disabling interrupts. If task (or ISR) occupies resource which is referenced by ISR of priority 'P' then all ISRs with priority equal or lower than 'P' are disabled and task rescheduling is disabled. Interrupts are reenabled and task rescheduling is reenabled after releasing the resource.

ISR Stack

The purpose of the ISR stack is to save memory. Since interrupts can occur while any task is executed and can be nested it means that every task stack must be big enough to store several interrupt stack frames (in addition to task and ISR(s) needs for local variables,

function calls, etc.). To avoid this overhead, the separate ISR stack is used in the OSEKturbo OS. Switching to this stack is performed by the OS at the beginning of ISR category 2. This stack is used only by ISRs of category 2. If nested interrupts occur after the stack has been switched, they will continue to use this stack. After completion of ISR category 2 OS switches back from ISR stack.

If BCC1 class, ISRs use common single stack.

The *interrupt stack frame* usually consists of the CPU registers, and optionally some compiler-dependent 'virtual' registers. The CPU registers are pushed onto the stack under hardware or software control. In the latter case the compiler generates a stack frame by means of adding special sequences of machine instructions before the first statement in the function.

Most compilers use function modifiers (like 'interrupt') to generate stack frames. In turn, the *ISR* keyword, specified in OSEK (see "Conventions"), is a macro for this modifier.

Interrupt Dispatcher

OSEKturbo OS/MPC5xx provides interrupt dispatcher for the external interrupt distinction. There is an *InterruptDispatcher* attribute which defines the type of OS interrupt dispatcher. It may have value of *None or OneLevel* or *MultiLevel*; for MPC565/563/561 it also can have a value *Enhanced.* In order to use Interrupt Dispatcher the user should set up interrupt level into control register of interrupt source for each external interrupt. For example, this initialization might be done before StartupOS call or from other part of user's code according to design. Besides, for each external interrupt it is necessary to define an ISR object in the OIL file and assign the *PRIORITY* attribute value according to the interrupt level set up in the controls register of the interrupt.

More details about the interrupt dispatcher see "Interrupt Related Properties" and "Interrupt Dispatcher". For more detailed information about MPC interrupt controller see MPC User's Manuals.

Programming Issues

Configuration Options

- *InterruptDispatcher*
  Specifies mode of interrupt dispatcher.

Run-time Services

OSEK OS provides the set of services for interrupt management. These services are shown in the Table 5.1.

Table 5.1   Interrupt Management Services in OSEK OS

Service Name	Description
DisableAllInterrupts	Disable all interrupts, does not allow nesting
EnableAllInterrupts	Restore state of interrupts saved by DisableAllInterrupts service
SuspendAllInterrupts	Disable all interrupts, allows nesting
ResumeAllInterrupts	Restore state of interrupts saved by SuspendAllInterrupts service
SuspendOSInterrupts	Disable interrupts of category 2
ResumeOSInterrupts	Restore state of interrupts saved by SuspendOSInterrupts service

Not all OS services may be used inside ISRs. All OS services that are allowed for use in the ISRs of category 2 are listed in the Table 5.2.

Table 5.2   Services allowed for use in ISRs of category 2

Service Name	Description
ActivateTask	Activates the specified task (puts it into the *ready* state)
GetTaskId	Gets reference to a task
GetTaskState	Gets state of the task
GetResource	Occupies a resource
ReleaseResource	Releases a resource
SetEvent	Sets event for the task
GetEvent	Gets event of the task

Table 5.2   Services allowed for use in ISRs of category 2

Service Name	Description
CounterTrigger[a]	Increments a counter value and process attached alarms
GetAlarmBase	Gets alarm base characteristics
GetAlarm	Gets value in ticks before the alarm expires
SetRelAlarm	Sets relative alarm
SetAbsAlarm	Sets absolute alarm
CancelAlarm	Cancels alarm
SendMessage	Sends an unqueued message in *WithCopy* configuration to the specified task
ReceiveMessage	Delivers data of an unqueued message in *WithCopy* configuration to the application message copy
GetRunningStackUsage[a]	Gets amount of stack used by running task
GetStackUsage[a]	Gets amount of stack used by certain task
GetTimeStamp[a]	Gets current value of system counter

[a].   This service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification.

Conventions

Within the application, an Interrupt Service Routine should be defined according to the following pattern:

```
ISR( <ISRName> )[1]
{
...
}
```

The keyword *ISR* is the macro for compiler specific interrupt function modifier, which is used to generate valid code to enter and exit ISR.

---

[1].   OSEK/VDX does not specifies using of keyword *ISR* for ISRs of category 1, it is OSEKturbo specific for this category.

The constructional statement *DeclareISR* declares ISR function. It may be useful in vector file if it does not include OS configuration file.

DeclareISR looks like the following:
```
DeclareISR( <name of ISR> );
```

The same name shall be used for corresponding ISR object definition (see "ISR Definition") and for interrupt vector definition (see "Interrupt Vector Table").

ISR Definition

To define common ISR parameters like ISR stack size the corresponding OS properties should be specified in the configuration file.

Definition of Interrupt related properties looks like:

```
OS <name> {
  .....
  IsrStackSize = 256;
};
```

Definition of specific ISR object looks like:

```
ISR Handler {
  PRIORITY = 8;
  CATEGORY = 2;
  RESOURCE = ISRresource;
  ACCESSOR = RECEIVED {
    MESSAGE = messageA;
    ACCESSNAME = myBuffer;
  };
};
```

See "ISR Definition" for details.

Interrupt Vector Definition

To provide starting ISR routine during interrupt at run-time the interrupt vector shall be defined and added to interrupt vector table (see "Interrupt Vector Table").

6

Resource Management

This chapter describes resource management and task coordination by resources.

This chapter consists of the following sections:

- General
- Access to Resources
- Programming Issues

General

The resource management is used to coordinate concurrent access of several tasks or/and ISRs to shared resources, e.g. management entities (scheduler), program sequences (critical sections), memory or hardware areas. In the OSEKturbo OS the resource management is provided in all Conformance Classes.[1]

The resource management ensures that

- two modules (tasks or ISRs) cannot occupy the same resource at the same time,
- priority inversion cannot arise while resources are used,
- deadlocks do not occur due to the use of these resources,
- access to resources never results in *waiting* state.

The functionality of the resource management is only required in the following cases:

- full-preemptable tasks,
- non-preemptable tasks, if the user intends to have the application code executed under other scheduling policies too.

[1.] This is the OSEKturbo extension of OSEK OS, which fully supports resources only in BCC2 and ECC conformance classes.

• resource sharing between tasks and/or ISRs.

Resources cannot be occupied by more than one task or ISR at a time. The resource which is now occupied by a task or ISR must be released before another task or ISR can get it. The OSEK operating system ensures that tasks are only switched from *ready* to *running* state if all the resources which might be occupied by that task during its execution have been released. The OSEK operating system ensures that an ISR is enabled if all the resources which might be occupied by that ISR during its execution have been released. Consequently, no situation occurs in which a task or an ISR tries to access an occupied resource. A special mechanism is used by the OSEK Operating System to provide such behavior, see "Priority Ceiling Protocol" for details**.**

In case of multiple resource occupation, the task or ISR must request and release the resources following the LIFO principle (stack). For example, if the task needs to get the communication hardware and then the scheduler to avoid possible preempts, the following code may be used:

```
GetResource( SCI_res );        /* occupy the SCI resource */
...                            /* user's code */
GetResource( RES_SCHEDULER ); /* occupy the scheduler resource */
...                            /* user's code */
ReleaseResource( RES_SCHEDULER ); /* release the scheduler */
ReleaseResource( SCI_res );     /* release the SCI resource */
```

The OSEK OS resource management allows the user to prevent such situations as priority inversion and deadlocks which are the typical problems of common synchronization mechanisms in real-time applications (e.g., semaphores).

It is not allowed to occupy RES_SCHEDULER resource in the ISR.

Access to Resources

Before they can be used, the resources must be defined by the user at the system configuration stage through the *RESOURCE* definition, see "Resource Definition". The resource must be referenced in all the TASKs and ISRs which can occupy it. (A special

resource RES_SCHEDULER is referenced and can be used by any TASK by default.) After that the task or ISR can occupy and release the resource using the *GetResource* and *ReleaseResource* services. While the resource is occupied, i.e. while the code between these services is executed, this resource cannot be occupied by another task or ISR.

In the OSEK Operating System the resources are ranked by priority. The priority which is statically assigned to each resource is called *Ceiling Priority*. The resource priority is calculated automatically during the system generation. It is possible to have resources of the same priority, but the Ceiling Priority of the resource is higher or equal to the highest task or ISR priority with access to this resource.

Restrictions when using resources

TerminateTask, ChainTask, Schedule and *WaitEvent* must not be called while a resource is occupied. The interrupt service routine must not be completed with the resource occupied.

The OSEK strictly forbids the nested access to the same resource. In the rare cases when the nested access is required, it is recommended to use a second resource with the same behaviour as the first resource. The OIL language especially supports the definition of resources with identical behaviour (so-called 'linked resources').

Priority Ceiling Protocol

The Priority Ceiling Protocol is implemented in the OSEK Operating System as a resource management discipline.

The priority ceiling protocol elevates the task or ISR requesting a resource to a resource priority level. This priority can be simply calculated during the system generation. As shown in "General" the Ceiling Priority is:

- Higher or equal to the highest task or ISR priority with access to this resource (task T1);
- Lower than the priority of those tasks or ISR which priority is higher than the one of task T1.

Note that all ISR priorities are higher than any task and scheduler priorities.

For More Information: www.freescale.com

When a task or ISR occupies a resource, the system temporarily changes its priority. It is automatically set to the Ceiling Priority by the resource management. Any other task or ISR which might occupy the same resource does not enter *running* state (ISR stays pending and cannot start) due to its lower or equal priority. If the resource occupied by the task or ISR is released, the task (ISR) returns to its former priority level. Other tasks which might occupy this resource can now enter *running* state (ISR can start).

Hardware interrupt levels and interrupt flags are used by resources which can be occupied in the ISR. When such a resource is occupied by a task or ISR, the interrupts of the corresponding priority are disabled and the OSEK OS scheduler is switched off. Therefore, the *running* task can not be switched to *ready* state while such a resource is occupied. Releasing the resource causes enabling interrupts of the corresponding level and switching on the OSEK OS scheduler.

The example shown in <u>Figure 6.1</u> illustrates the mechanism of the Priority Ceiling Protocol.

Figure 6.1   Priority Ceiling Protocol
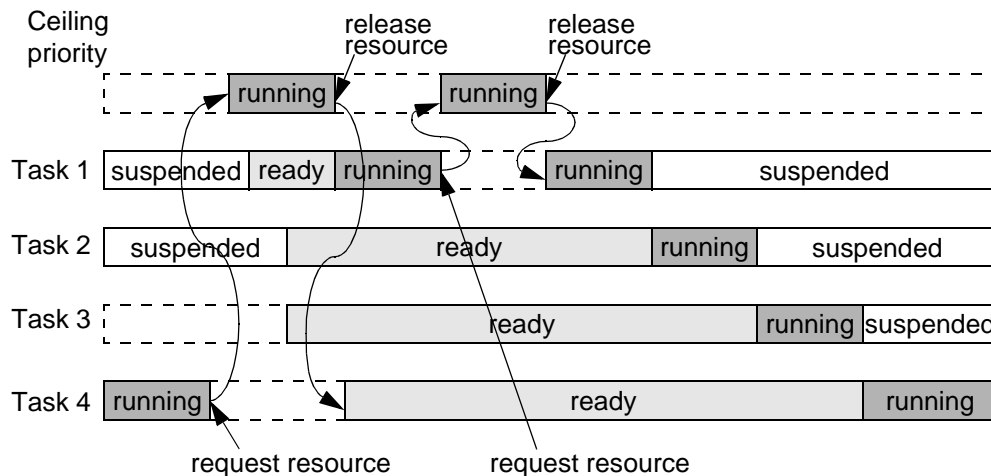
In the figure above Task 1 has the highest priority and Task 4 has the lowest priority. The resource has a priority greater than or equal to the Task 1 priority. When Task 4 occupies the resource, it gets a priority not less than the Task 1 has, therefore it cannot be preempted by *ready* Task 1 until it releases the resource. As soon as

the resource is released, Task 4 is returned to its low priority and becomes *ready*, and Task 1 becomes the *running* task. When Task 1, in turn, occupies the resource, its priority is also changed to the Ceiling Priority.

Scheduler as a Resource

The OSEK operating system treats the scheduler as a specific resource which is accessible to all tasks. Therefore, a standard resource with the predefined identifier *RES_SCHEDULER* is generated, and it is supported in all Conformance Classes. If a task calls the services *GetResource* or *ReleaseResource* with this identifier as a parameter, the task will occupy or release the scheduler in the manner of a simple resource. See the code example in "General".

If a task wants to protect itself against preemptions by all other tasks, it can occupy the scheduler exclusively. When it is occupied, interrupts are received and processed normally. However, it prevents the tasks rescheduling.

NOTE   If a task gets the scheduler as a resource, it must release it before the point of rescheduling!

Reference to RES_SCHEDULER from TASK object is optional. The user may define the resource with the name RES_SCHEDULER in the OIL file but this resource will have the maximal (scheduler) prioriry regardless of which tasks have the reference to it. The RES_SCHEDULER is referenced and can be used from any task by default. The RES_SCHEDULER resource cannot be occupied from the ISR.

Internal resources

The internal resources are the resources which are not visible to the user and therefore they can not be addressed by the system functions GetResource and ReleaseResource. Instead, they are managed strictly internally within a clearly defined set of the system functions. Besides, the behaviour of the internal resources is exactly the same as the behaviour of standard resources (the priority ceiling protocol etc.). At most one internal resource can be assigned

to a task during the system generation. If an internal resource is assigned to a task, the internal resource is managed as follows:

- The resource is automatically taken when the task enters running state, except when it has already taken the resource. As a result, the priority of the task is automatically changed to the ceiling priority of the internal resource.

- At the points of rescheduling, defined in chapter "Scheduling Policy", the resource is automatically released.

The tasks which have the same internal resource assigned form a group of tasks. The tasks within a certain group behave like the non preemptable tasks - they can not preempt each other; while the tasks with the priority higher than the highest priority within the group preempt the tasks within the group.

The non preemptable tasks may be considered as a special group with an internal resource of the same priority as the RES_SCHEDULER priority (chapter "Non-preemptive Scheduling"). The internal resources can be used in all cases when it is necessary to avoid unwanted rescheduling within a group of tasks. More than one internal resource can be defined in a system thus defining more then one group of tasks.

The general restriction on some system calls that they must not be called with the resources occupied (see "Restrictions when using resources") is not applied to the internal resources, as the internal resources are handled within those calls.

The tasks which have the same assigned internal resource cover a certain range of priorities. It is possible to have the tasks which do not use this internal resource in the same priority range, but these tasks will not belong to the group. So they are preemptable by the tasks of the group.

Programming Issues

Configuration Option

The following system configuration option is intended to decrease the amount of RAM and ROM used by the OS:

- *ResourceScheduler*
  If this option is set to FALSE, *RES_SCHEDULER* is not
  supported by the OS.

Data Types

The OSEK OS determines the following data type for the resource
management:

- *ResourceType*
  The abstract data type for referencing a resource.

The only data type must be used for operations with resources.

Run-time Services

The OSEK OS provides a set of services for the user to manage
resources. Detailed descriptions of these services are provided in
"Resource Management Services". Below is only a brief list.

Table 6.1   Resource Management Run-time Services

Service Name	Description
GetResource	This call serves to occupy the resource (to enter critical section of the code, assigned to the resource)
ReleaseResource	Releases the resource assigned to the critical section (to leave the critical section)

Resource Definition

To define a resource, the following definition statement should be
specified in the application configuration file:

```
RESOURCE ResourceName {
  RESOURCEPROPERTY = STANDARD;
  };
```

For more details see "Resource Definition".

The declaration statement *DeclareResource* may be used for
compatibility with previous OSEK versions. It may be omitted in
the application code.

DeclareResource is as follows:

```
DeclareResource( <ResourceName> );
```

7

Counters and Alarms

This chapter describes usage of these control mechanisms in OSEK OS.

This chapter consists of the following sections:

- General
- Counters
- Alarms
- Alarm Callback
- TimeScale
- Programming Issues

General

The OSEK operating system comprises a two level concept to make use of recurring occasions like periodic interrupt of timers, interrupt of the sensors on rotating angles, or any recurring software occasions. To manage such a situation, counters and alarms are provided by the OSEKturbo OS. Additionally OSEKturbo OS provides a TimeScale mechanism for fast tasks activations in accordance with statically defined schedule. The recurring occasions (sources) can be registered by counters. Based on counters, the OSEK OS offers an alarm mechanism to the application software. Counters and alarms are provided by the OSEK OS in all Conformance Classes. Counter concept and counter management system services are based on the *OSEK Operating System, Concept v.1.00, September 1995* and *OSEK Operating System, Application Program Interface, v.1.00, 11 September 1995* documents.

OSEKturbo OS provides two types of counters: software (SW) counters, that are the same as in previous versions of OSEK and hardware (HW) counters. HW counters allow more precise timing while decreases system overhead time because the timer interrupts are occurs only when the alarm(s) attached to the counter expiries.

Figure 7.1   Counters and Alarms
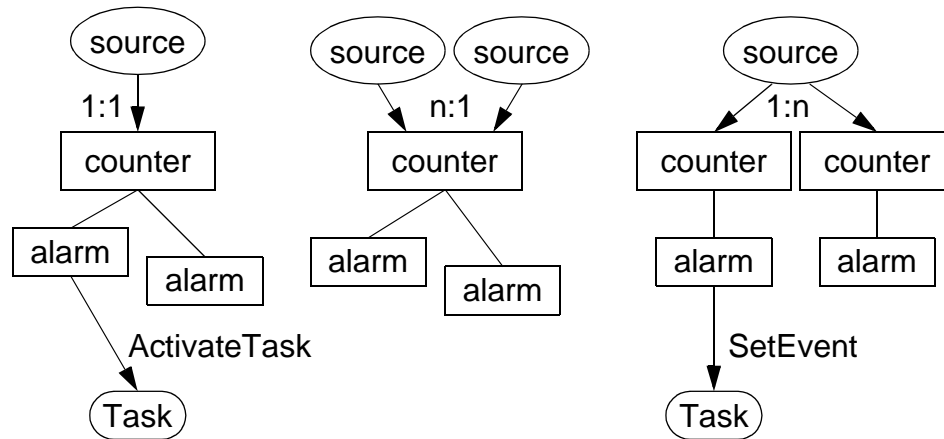
Counters

Any occasion in the system can be linked with a counter, so that when the occasion has occurred, the counter value is changed. A counter is identified in the system via its symbolic name, which is assigned to the counter statically at the configuration stage.

A counter is represented by a current counter value and some counter specific parameters: *counter initial value*, *conversion constant* and *maximum allowed counter value*. They are defined by the user. The latter two parameters are constants and they are defined at system generation time. The counter initial value is the dynamic parameter. The user can initialize the counter with this value and thereafter on task or on interrupt level advance it using the system service *CounterTrigger*.

The HW counters may use only *System* and *Second* Timers and has a maximum allowed value 0xFFFFFFFF. Only time base reference registers TBREF0 (TB0) and TBREF1 (TB1) may be used with HW counters. The source(s) for HW counters is hardware itself. For HW counters one tick of hardware timer is equivalent to a *Period* for SW counter, thus enabling more precise timing while keeping system overhead on interrupt processing low because timer interrupts are raised only when alarms attached to this counter are expired.

The maximum allowed counter value specifies the number after which the counter rolls over. After a counter reaches its maximum allowed possible value (or rolls over the predefined size - 32 bits), it starts counting again from zero.

The conversion constant can be used to convert the counter value into an appropriate user specific unit of measurement, e.g. seconds for timers, angular degrees for rotating axles. The conversion is done by the user's code and this parameter can be treated as a counter-specific reference value.

The operating system provides the standard service *GetCounterInfo* to read these counter specific values. Also the service *GetCounterValue* is designed to read the current counter value.

OSEKturbo OS provides two timers (the internal system clocks): a *system timer* and a *second timer*. The timers are not defined in OSEK OS specifications starting from OSEK OS v.2.0 specification. This is OSEKturbo extension of OSEK OS. User can turn on or turn off the system timer using the *SysTimer* attribute and the second timer using the *SecondTimer* attribute. The timer can be assigned to a standard counter with the following additions:

- special constants are defined to describe counter parameters and to decrease access time;
- the user defines the source of hardware interrupts for the counter attached to the timer.

In the system definition statement for the system (second) timer the user should define one of possible hardware interrupt sources. Parameters to tune the hardware can be also defined by the user in this statement. This possibility allows the user to exactly tune the system (see "PowerPC Platform-Specific Features" for details).

While hardware related parameters are defined, the code to initialize the system (second) timer hardware and the interrupt handler are automatically provided for the user as a part of OSEK OS. The handler is an ISR category 2 but it is not needed to define the ISR in OIL file. In that case the user does not have to care about handling of this interrupt and he/she can not change the provided code.

Software Counters may be triggered from user defined ISR(s). Hardware interrupts which are used to trigger counters have to be handled in usual manner. To perform any actions with the counter the application software processing the occasion should call the *CounterTrigger* system service. It is not allowed to use *CounterTrigger* in ISR category 1 (see section "ISR Categories").

NOTE CounterTrigger service is not defined starting from OSEK OS v.2.0 specifications, it is defined in the OSEK OS v.1.0 specification.

The user is free to assign one source exactly to one counter (1:1 relationship), several sources to one counter (n:1 relationship), or one source to several counters (1:n relationship), see Figure 7.1. Meaning that it is possible to advance the same counter in different software routines.

Alarms

The alarm management is built on top of the counter management. The alarm management allows the user to link task activation or event setting or a call to callback function to a certain counter value. These *alarms* can be defined as either single (one-shoot) or cyclic alarms.

The OSEK OS allows the user to set alarms (relative or absolute), cancel alarms and read information out of alarms by means of system services. Alarm is referenced via its symbolic name which is assigned to the alarm statically at the configuration stage.

Examples of possible alarm usage are:

– 'Activate a certain task, after the counter has been advanced 60 times', or

– 'Set a certain event, after the counter has reached a value of 90'.

The counter addressed in the first example might be derived from a timer which is advanced every second. The task in the example is then activated every minute. The counter addressed in the second example might be derived from a rotating axle. The event is set on a 90 degree angle.

The OSEK OS takes care of the necessary actions of managing alarms when a counter is advanced.

Alarms are defined statically as with all other system resources. The assignment of alarms to counters, as well as the action to be performed when an alarm expires (task and event), are also defined statically. An application can use an alarm after it has been defined and assigned to a counter. Alarms may be either in the stop state or running state. To run an alarm, the special system services are used, which set dynamic alarm parameters to start it.

Dynamic alarm parameters are:

- the counter value when an alarm has to expire.
- the cycle value for cyclic alarms.

An alarm can be started at any moment by means of system services *SetAbsAlarm* or *SetRelAlarm.* An alarm will expire (and predefined actions will take place) when a specified counter value is reached. This counter value can be defined relative to the actual counter value or as an absolute value. The difference between relative and absolute alarms is the following:

- Relative alarm expires when the specified number of counter ticks has elapsed, starting from the current counter value at the moment the alarm was set.
- Absolute alarm expires when the counter reaches the specified number of ticks, starting from zero counter value no matter which value the counter had at the moment the alarm was set. If the specified number of ticks is less than the current counter value, the counter will roll over and count until the specified value. If the specified value is greater than the current value, the alarm will expire just after the counter reaches the desired number. This is illustrated by Figure 7.2. In the latter case, the total time until the alarm expires is the sum of T_1 and T_2.

For More Information: www.freescale.com

Figure 7.2   Two Cases for the Absolute Alarm
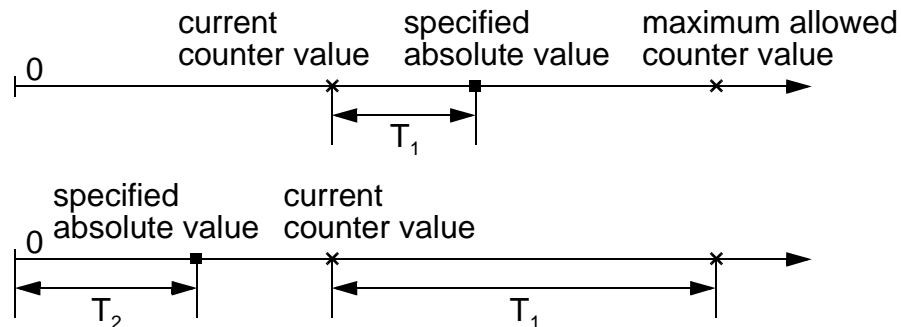
If a cycle value is specified for the alarm, it is logged on again immediately after expiry with this relative value. Specified actions (task activation or event setting) will occur when the counter counts this number of ticks, starting from the current value. This behavior of the cyclic alarm is the same both for relative and absolute alarms. If the cycle value is not specified (it equals zero) the alarm is considered as a single one.

WARNING!   It is not recomended to use values of cycle and/or increment parameters close to 0xFFFFFFFF (hardware counter *MAXALLOWEDVALUE*) for Alarms configured on a Hardware Timer. The difference between *MAXALLOWEDVALUE* and this values should be greater than interrupt latency of the system (time spent in the longest ISR).

Alarm Callback

User can define alarm callback function for each alarm. The function is placed in user application and its name added to ALARM object definition as value of *ALARMCALLBACKNAME* attribute. The alarm callback is the usual user's function. It can have neither parameter(s) nor return value.

Only the *SuspendAllInterrupts* and *ResumeAllInterrupts* services may be used within alarm callback. No other OS services shall be called.

The callback function shall have next definition:

```
ALARMCALLBACK(CallbackName)
{
   /* user application code */
}
```

TimeScale

OSEKturbo OS provides a special feature for fast tasks activations in accordance with statically defined schedule named *TimeScale*[1]. User can define in a configuration file a sequence of tasks to be cyclically activated at predefined time points. *TimeScale* always uses the System Timer which must be defined as a *HWCOUNTER* in this case. Only one task may be activated at each *Step* of *TimeScale* but the *StepTime* of any *Step* may be set to '0' to achieve a simultaneous activation of two or more tasks. *TimeScale* has a better performance than cyclic *Alarms* because of simplified algorithm and reduced system overhead. System Timer can not be used for *TimeScale* and *Alarms* simultaneously.

Programming Issues

Configuration Options

The following system configuration options affect the counter and alarm management:

- *SysTimer*
  If this option is turned on the System Timer is used.

- *SecondTimer*
  If this option is turned on the Second Timer is used.

- *TimeScale*
  If this option is turned on the Time Scale is used.

[1.] TimeScale is OSEKturbo extension of OSEK OS.

Freescale Semiconductor, Inc.

Counters and Alarms
Programming Issues

Data Types

The following data types are established by OSEK OS to work with counters:

- *CtrRefType*
  The data type references a counter

- *TickType*
  The data type represents a counter value in system ticks

- *TickRefType*
  The data type references data corresponding to the data type *TickType.* Reference to *TickType* variable can be used instead of *TickRefType* variable.

- *CtrInfoType*
  This data type represents a structure for storage of counter characteristics. This structure has the following fields:

  - *maxallowedvalue*
    maximum possible allowed count value;

  - *ticksperbase*
    number of ticks required to reach a counter-specific significant unit;

  - *mincycle*
    minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status).

    All fields have the data type *TickType.* The following code may illustrate usage of this data type:

```
CtrInfoType CntData;
TickType maxV, minC, cons;
GetCounterInfo( CntID, &CntData );
maxV = CntData.maxallowedvalue;
minC = CntData.ticksperbase;
cons = CntData.mincycle;
```

- *CtrInfoRefType*
  This data type references data corresponding to the data type *CtrInfoType.* Reference to *CtrInfoType* variable can be used instead of *CtrInfoRefType* variable

TR–72   OSEKturbo OS/MPC5xx

For More Information: www.freescale.com

NOTE CtrRefType, CtrInfoType and *CtrInfoRefType* data types are not defined in *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

The following data type is established by OSEK OS to work with alarms:

- *AlarmBaseType*
  This data type represents a structure for storage of alarm characteristics. It is the same as *CtrInfoType*;

- *AlarmBaseRefType*
  This data type references data corresponding to the data type *AlarmBaseType*;

- *AlarmType*
  The data type represents an alarm element.

Counters and Alarm Generation

To generate a counter in an application, the *COUNTER* definition is used, it looks like the following:

```
COUNTER CounterName {
  MINCYCLE = 5;
  MAXALLOWEDVALUE = 1000;
  TICKSPERBASE = 10;
};
```

To define system or second timer hardware-specific parameters, the following properties should be defined in the OS definition statement:

```
OS <name> {
  ...
  SysTimer = HWCOUNTER {
    COUNTER = <CounterName>;
    ISRPRIORITY = <priority>;
    TimerHardware = <TypeOfTimer> {
      Prescaler = OS {
        Value = <PrescalerValue>;
      };
    };
```

```
  };
  SecondTimer = SWCOUNTER {
    COUNTER = <CounterName>;
    ISRPRIORITY = <priority>;
    TimerHardware = <TypeOfTimer> {
      Prescaler = OS {
        Value = <PrescalerValue>;
      };
      TimerModuloValue = <TimerModuloValue>;
    };
  };
  TimeScale = TRUE {
    TimeUnit = <ticks, ns, us, ms>;
    ScalePeriod = <timescale period>;
    Step = SET {
          StepNumber = <1,2,3....>;
          StepTime = <time to next step>;
          TASK = <ID of task to activate>;
      };
    ...
  };

  ...
};
```

The system timer hardware parameters are described in detail in the section "CPU Related Attributes".

To generate an alarm in an application, the *ALARM* definition is used, it looks like the following:

```
ALARM AlarmName {
  COUNTER = CounterName;
  AUTOSTART = FALSE;
  ACTION = SETEVENT {
    TASK = TaskName;
    EVENT = EventName;
  };
};
```

Detailed counter and alarm generation statements are described in "Counter Definition" and "Alarm Definition".

The declaration statements *DeclareCounter* and *DeclareAlarm* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

```
DeclareCounter( <CounterName> );
DeclareAlarm( <AlarmName> );
```

Run-time Services

OSEK OS grants a set of services for the user to manage counters and alarms. Detailed descriptions of these services are provided in "System Services". Here only a brief list is given.

Table 7.1   Counter and Alarm Management Run-time Services

Service Name	Description
InitCounter	Sets the initial value of the counter
CounterTrigger	Increments the counter value and process attached alarms
GetCounterValue	Gets the counter current value
GetCounterInfo	Gets counter parameters
SetRelAlarm	Sets the alarm with a relative start value
SetAbsAlarm	Sets the alarm with an absolute start value
CancelAlarm	Cancels the alarm: the alarm is transferred into the STOP state
GetAlarm	Gets the time left before the alarm expires
StartTimeScale	Starts TimeScale processing
StopTimeScale	Cancels TimeScale processing
GetAlarmBase	Gets alarm base (attached counter) parameters

NOTE   InitCounter, CounterTrigger, GetCounterValue, GetCounterInfo, StartTimeScale, and StopTimeScale services are not defined in *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Examples of the run-time service usage are provided in <u>"System Services"</u>.

Constants

For all counters, the following constants are defined:

- *OSMAXALLOWEDVALUE_cname*
  Maximum possible allowed value of counter <cname> in ticks.

- *OSTICKSPERBASE_cname*
  Number of ticks required to reach a specific unit of counter <cname>.

- *OSMINCYCLE_cname*
  Minimum allowed number of ticks for a cyclic alarm of counter <cname>. This constant is not defined in *STANDARD* status

For the counters attached to *SysTimer* and *SecondTimer* special constants are provided by the operating system:

- *OSMAXALLOWEDVALUE*
  maximum possible allowed value of the system timer in ticks;

- *OSMAXALLOWEDVALUE2*
  maximum possible allowed value of the second timer in ticks;

- *OSTICKSPERBASE*
  number of ticks that are required to reach a counter-specific value in the system counter;

- *OSTICKSPERBASE2*
  number of ticks that are required to reach a counter-specific value in the second counter;

- *OSTICKDURATION*
  duration of a tick of the system counter in nanoseconds;

- *OSTICKDURATION2*
  duration of a tick of the second counter in nanoseconds;

- *OSMINCYCLE*
  minimum allowed number of ticks for a cyclic alarm attached to the system counter (only for system with Extended Status);

- *OSMINCYCLE2*
  minimum allowed number of ticks for a cyclic alarm attached to the second counter (only for system with Extended Status).

NOTE   *OSMAXALLOWEDVALUE2* , *OSTICKSPERBASE2*, *OSTICKDURATION2*, and *OSMINCYCLE2* constants are not

For More Information: www.freescale.com

defined in the OSEK/VDX Operating System specification. These constants are OSEKturbo extension of OSEK OS.

8

Events

This chapter is devoted to event management and task coordination by events.

This chapter consists of the following sections:

- General
- Events and Scheduling
- Programming Issues

General

Within the OSEK operating system, tasks and ISRs can be synchronized via occupation of a resource (see "Resource Management"). Another means of synchronization is the event mechanism, which is provided for Extended Tasks only. Events are the only mechanism allowing a task to enter the *waiting* state.

An *event* is a synchronization object managed by the OSEK Operating System. The interpretation of the event is up to the user. Examples are: the signalling of a timer's expiry, the availability of data, the receipt of a message, etc.

Within the operating system, events are not independent objects, but allocated to Extended Tasks. Each event is represented by a bit in event masks which belongs to Extended Tasks. Maximum number of events for Extended Task is 32. Each Extended Task has the mask of a "set" events and the mask of events the task is waiting for ("wait" mask). When the Extended Task is activated all its events are cleared.

An Extended Task can wait for several events simultaneously and setting at least one of them causes the task to be transferred into the *ready* state. When a task wants to wait for one event or several ones, the corresponding bits in its "'wait" event mask are set by the system service *WaitEvent* which is designed to force a task to wait

---

for an event. When another task sets an event, it sets the specified bits of the "set" event mask and if some bits in both "wait" and "set" masks are the same the task is transferred into the *ready* state. The task can clear its own events in the "set" event mask using *ClearEvent* service.

All tasks can set any events of any Extended Task. Only the appropriate Extended Task (the owner of the particular event mask) is able to clear events and to wait for the setting (receipt) of events. Basic Tasks must not use the operating system services for clearing events or waiting for them.

An alarm can also be set for an Extended Task, which in turn sets an event at a certain time. Thus, the Extended Task can delay itself (see example in "Examples of Using Events").

It is not possible for an interrupt service routine or a Basic Task to wait for an event, since the receiver of an event is an Extended Task in any case. On the other hand, any task or ISR can set an event for an Extended Task, and thus inform the appropriate Extended Task about any status change via this event.

Events and Scheduling

An event is an exclusive signal which is assigned to an Extended Task. For the scheduler, events are the criteria for the transition of Extended Tasks from the *waiting* state into the *ready* state. The operating system provides services for setting, clearing and interrogation of events, and for waiting for events to occur.

Extended Tasks are in the *waiting* state if an event for which the task is waiting has not occurred. If an Extended Task tries to wait for an event and this event has already occurred, the task remains in the *running* state.

Figure 8.1 illustrates the procedures which are effected by setting an event: Extended Task 1 (with higher priority) waits for an event. Extended Task 2 sets this event for Extended Task 1. The scheduler is activated. Subsequently, Task 1 is transferred from the *waiting* state into the *ready* state. Due to the higher priority of Tasks 1 this results in a task switch, Task 2 being preempted by Task 1. Task 1

resets the event. Thereafter Task 1 waits for this event again and the scheduler continues execution of Task 2.

Figure 8.1   Synchronization by Events for Full-preemptive Scheduling
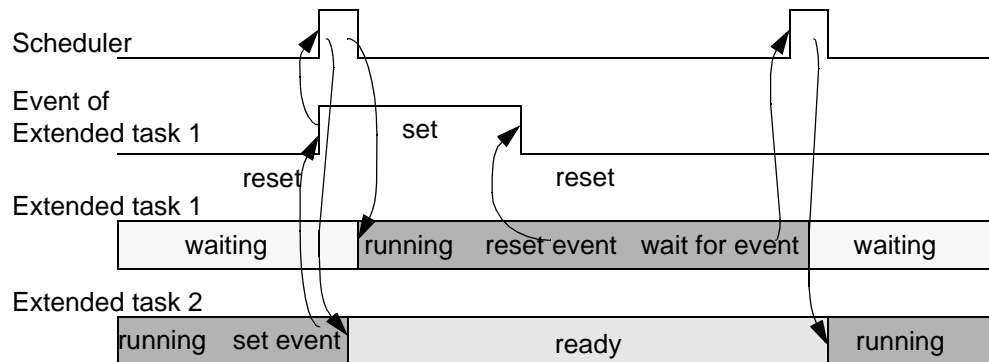
If non-preemptive scheduling is supposed, rescheduling does not take place immediately after the event has been set, as shown in Figure 8.2.

Figure 8.2   Synchronization by Events for Non-preemptive Scheduling
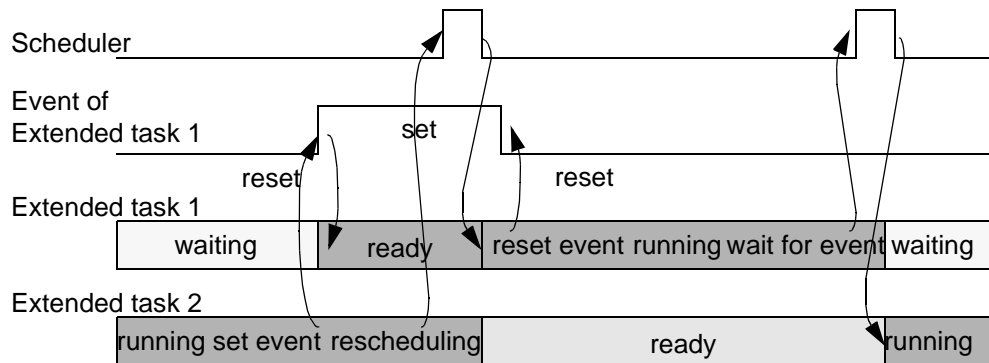
Programming Issues

Configuration Options

There are no any system configuration options controlling event management in the system.

Data Types

The OSEK Operating System establishes the following data types for the event management:

- *EventMaskType*
  The data type of the event mask;

- *EventMaskRefType*
  The data type to refer to an event mask. Reference to *EventMaskType* variable can be used instead of *EventMaskRefType* variable.

The only data types must be used for operations with events.

Events Definition

To generate an event in an application the *EVENT* definition is used, it looks like the following:

```
EVENT EventName {
   MASK = 0x01;
};
```

The declaration statement *DeclareEvent* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

DeclareEvent looks like the following:

```
DeclareEvent( <Event> );
```

Some task event which used by the task as internal flags can be undefined in OIL file. But it is strictly recommended to define all events and reference them in TASK object. Missing of an event in

For More Information: www.freescale.com

OIL file can lead to wrong mask assignment. If task has no references to events the task is considered as basic task.

Run-time Services

OSEK OS grants a set of services for the user to manage events. A detailed description of these services is provided in "Event Management Services". Here only a brief list is given.

Table 8.1   Event Management Run-time Services

Service Name	Description
SetEvent	Sets events of the given task according to the event mask
ClearEvent	Clears events of the calling task according to the event mask
GetEvent	Gets the current event setting of the given task
WaitEvent	Transfers the calling task into the waiting state until specified events are set

Examples of the run-time services usage are provided in section "Event Management Services".

9

Communication

This chapter describes message concept in OSEK and their usage.

This chapter consists of the following sections:

- Message Concept
- Unqueued Messages
- Queued Messages
- Data Consistency
- Programming Issues

Message Concept

In the OSEK Operating System communication between application tasks and/or ISRs takes place via messages. Communication concept and message management system services are based on the *OSEK/VDX Communication, v.2.2.2, 18 December 2000* specification. OSEKturbo OS supports CCCB (Communication Conformance Class B) which includes support of internal *Unqueued* and *Queued Messages.*

An Unqueued Message represents the current value of a system variable, e.g. engine temperature, wheel speed, etc. Unqueued Messages are not buffered but overwritten with their actual values. The receive operation reads the Unqueued Message value. Thereby the message data is not consumed.

By contrast, Queued Messages are stored in a FIFO buffer and they are read by the application in the order they arrived. A particular Queued Message can therefore be read only once, as the read operation removes it from the queue. A Queued Message would normally be used to track changes of state within a system, where it is important that receiver maintains synchronisation of state information with the sender.

In OSEK OS message objects are referenced by tasks and ISRs via the unique identifiers defined by the user at the configuration stage.

The OSEK Operating System ensures data consistency of message data during task operation, uniform in all types of scheduling. The received unqueued message data remains unchanged until a further send operation is performed, unless the task or function using the data overwrites the data with a direct access operation.

As an option, *task activation, event signalling, flag* or *callback* function can be defined statically to be performed at message arrival to notify a task. Task activation or event signalling can be used to inform tasks that want to act immediately on new message information. There is no special operating system service to wait for messages, but the normal event mechanism is used. Only one notification method can be assigned for certain message.

OSEK OS communication services provide all means for internal message transfers. To transfer data over the network, the OSEK Communication System (COM) shall be used, which is designed to handle all other types of communication through the network. And OSEK OS communication services provide an interface for application tasks to exchange data. Thus, messages serve as interface for both internal and network communication. Uniform services with identical interfaces are offered (network transparency).

Unqueued Messages

Unqueued Messages represent the current value of a state variable. Tasks and ISRs have personal accessors to read or to write message. Accessor type is defined by SENT and RECEIVED values of ACCESSOR attribute respectively. Also a message can be accessed directly through the message buffer or indirectly through the message copy. Access through the message copy guarantees consistency of information in the message between adjacent operations of send/receive. This behavior is defined by WITHOUTCOPY attribute for each defined ACCESSOR. The send operation overwrites the current value of a message. The receive operation reads the current value of an Unqueued Message whereby the message data is not consumed. Allocation of memory for message copies depends on the *MessageCopyAllocation*[1] attribute.

It specifies whether SysGen will generate copies of messages in global memory or message copies are allocated in the application by the user. This is defined by OS and USER values respectively. The user can place copies of messages into the global or into the local memory of the functions. ISR accessors access the messages through the copies only.

The *SendMessage* and *ReceiveMessage* services ensure that the consistent writing and reading of message data within the send and receive operation (also in preemptive systems).

When an Unqueued Message is received with copy the information from the message buffer is copied to the message copy and further work with message information is performed using this copy.

When an Unqueued Message is sent with copy the information from the message copy is replicated to the message buffer and message initialization is performed using the message copy.

When an Unqueued Message is received without copy the information from the message buffer is accessed directly by the user application and consistency of the information is not guaranteed.

When an Unqueued Message is sent without copy the information from the message buffer is accessed directly by the user application and consistency of the information is not guaranteed.

The *ACCESSNAME* attribute defines the symbol that will be used in C code to access message data. *AccessNameRef* of *SendMessage/ReceiveMessage* services is a pointer to symbol specified in *ACCESSNAME* attribute. For example:

```
type _accessor;
_accessor.value1 = 0;
_accessor.value2 = 20;
SendMessage (MsgA, &_accessor);
```

where type is the value of CDATATYPE attribute for MsgA message object in OIL file. Then _accessor can be used as a pointer to the message body.

[1.] this attribute is OSEKturbo extension of the OSEK OS

1:N communication for Unqueued Messages does not have any difference from 1:1 communication, since any task can read the Unqueued Messages if its identifier is known.

Queued Messages

Queued Messages are stored in a FIFO buffer and they are read by the application in the order they arrived. Only tasks (not ISRs) have personal accessors to read or to write message. Accessor type is defined by SENT and RECEIVED values of ACCESSOR attribute respectively. The first message in the queue can be accessed through the message copy. ACCESSOR for an queued message cannot be defined with WITHOUTCOPY attribute equal to TRUE. The send operation adds value of a message to the end of the message queue. The receive operation reads the first value of a Queued Message in the queue and then removes it from the queue. Allocation of memory for message copies depends on the *MessageCopyAllocation* attribute. It specifies whether SysGen will generate copies of messages in the global memory or message copies are allocated in the application by the user. This is defined by OS and USER values respectively. The user can place copies of messages into the global or into the local memory of the functions.

The *SendMessage* and *ReceiveMessage* services ensure that the consistent writing and reading of message data within the send and receive operation (also in preemptive systems).

When a Queued Message is received the information from the first message in the message queue is copied to the message copy and further work with the message information is performed using this copy.

When a Queued Message is sent the information from the message copy is added to the end of the message queue.

The *ACCESSNAME* attribute defines the symbol that will be used in C code to access message data. *AccessNameRef* of *SendMessage/ReceiveMessage* services is a pointer to symbol specified in *ACCESSNAME* attribute. For example:

```
type _accessor;
_accessor.value1 = 0;
```

```
_accessor.value2 = 20;
SendMessage (MsgA, &_accessor);
```

where type is the value of CDATATYPE attribute for MsgA message object in OIL file. Then _accessor can be used as a pointer to the message body.

Data Consistency

Data consistency means that the content of a given application message correlates unambiguously to the operations performed onto the message by the application. This means that no unforeseen sequence of operations may alter the contents of the application message. Thus data consistency means that it can be guaranteed that a task can complete the calculation with the same data set. Data consistency is guaranteed using access with copy or using external synchronization mechanisms, e.g. events.

Programming Issues

Configuration Options

- *MessageCopyAllocation*
  This OSEKturbo specific attribute specifies whether System Generator generates copies of messages in global memory or message copies are allocated by the user.

Identifiers

The following names are used in the OSEK Operating System for work with messages:

- *SymbolicName*
  This is an unique name representing a message. It only can be used in conjunction with calls of the message service. A SymbolicName need not be a data type. Variables or constants of SymbolicName can be declared or used.

- *AccessName*
  This is a unique name defining access to a message object. Depending on the chosen configuration, a distinction is made between the following AccessName scheme:

WITHCOPY configuration:
A application variable exists as a copy of the message. The name of the variable is the AccessName. This variable contains a copy of the corresponding message object.

WITHOUTCOPY configuration:
The message object data is accessed via the AccessName. This AccessName is a static link: it refers directly to the message object data. The AccessName refers to the same data (RAM) as the message object.

- *AccessNameRef*
  This is the address of the message buffer or message copy.

- *FlagType*
  The abstract data type for flag identification.

Message Definition

Each message in an application is generated by means of using statements like the following:

```
MESSAGE MsgA {
  TYPE = UNQUEUED;
  CDATATYPE = "long int";
  ACTION = SETEVENT {
    TASK = task1;
    EVENT = eventC;
    };
};
```

In detail message configuration statements is described in <u>"Message Definition"</u>.

There are no constructional elements defined for messages.

Run-time Services

OSEKturbo OS grants a set of services for the user to manage messages. Detailed descriptions of these services are provided in

section "Communication Management Services". Here only a brief list is presented.

Table 9.1   Communication Management Run-time Services

Service Name	Description
SendMessage	Updates the message
ReceiveMessage	Gets the message
GetMessageStatus	Gets message status
GetMessageResource	Sets message status to BUSY
ReleaseMessageResource	Clears BUSY message status
ReadFlag	Returns the value of the specified notification flag
ResetFlag	Sets the specified notification flag to FALSE
InitCOM	Initializes COM hardware (empty in OSEKturbo)
CloseCOM	Resets COM hardware (empty in OSEKturbo)
StartCOM	Initializes COM internal data
StopCOM	Stops communications

Examples of the run-time services usage are provided in "System Services".

Callback Function

The user can define callback function for each message. The function is placed in user application and its name added to MESSAGE object definition as value of *CALLBACKNAME* attribute, in this case *ACTION* shall be defined as *CALLBACK*. The function is called when message arrives.

The callback function is usual user's function. It is executed on OS or ISR level and only *SendMessage* and *ReceiveMessage* services are allowed in it.

The callback function shall have next definition:

```
void <CallBackName> (void)
{
   /* user code */
}
```

For More Information: www.freescale.com

Usage of Messages

Messages are identified via a symbolic name. This identifier is used for references to the message when the system service is used.

If the message is used in the WITHCOPY configuration and the value of *MessageCopyAllocation* attribute is USER, then the variables to hold message's copies must be defined within the user's code by means of using the regular C-language definitions. If *MessageCopyAllocation* attribute is set to OS, then the variables to hold message's copies are defined by SysGen and the user shall not define this variables in code.

For example, if the user defines the message *MsgA* having type *int*, then user's code may access message using the following statements:

```
int _MsgA;

ReceiveMessage( MsgA, &_MsgA );
if( _MsgA == 2 ) { _MsgA = 1; }
SendMessage( MsgA, &_MsgA );
```

If the message is configured as *WITHOUTCOPY* property, then the pointer to the message body should be defined within the user's code using regular C-language statements. Again, because system generator creates *typedef* declaration for message item, it is recommended to use this declaration for definition of pointer, which is used to access message data.

For example, if the user defines the *WITHOUTCOPY* message `MsgB`, having the type `int` and ACCESSNAME MESSB, then user's code may access message using the following statements:

```
ReceiveMessage( MsgB, &MESSB );
if( MESSB == 2 ) { MESSB = 1; }
SendMessage( MsgB, &MESSB );
```

10

Error Handling and Special Routines

This chapter describes support provided to the user to debug an application and handle errors.

This chapter consists of the following sections:

- General
- Hook Routines
- Error Handling
- Start-up Routine
- Application Modes
- System Shutdown
- Programming Issues

General

The OSEK Operating System provides the user with tools for error handling and simple debugging at run time. These are special hook routines with names specified by OSEK OS that are to be written by the user. In this section, error handling at the system configuration stage is not considered; it is described in "System Objects Definition".

Hook Routines

The OSEK Operating System supports system specific *hook routines* to allow user-defined actions within the OS internal processing.

These hook routines in OSEK OS are:

- Called by the operating system, in a special context depending on the implementation of the operating system
- Can not be preempted by tasks
- not interrupted by category 2 interrupt routines
- Using an implementation-dependent calling interface
- Part of the operating system, but user defined
- Implemented by the user
- Standardized in interface per OSEK OS implementation, but not standardized in functionality (environment and behavior of the hook routine itself), therefore usually hook routines are not portable
- Only allowed to use a subset of API functions
- Optional

In the OSEK OS hook routines are intended for:

- System startup. The corresponding hook routine (*StartupHook*) is called after the operating system startup and before the scheduler is running
- Tracing or application dependent debugging purposes as well as user defined extensions of the context switch
- Error handling. The corresponding hook routine (*ErrorHook*) is called if a system call returns a value not equal to *E_OK*
- System shutdown. The corresponding hook routine (*ShutdownHook*) is called

Besides standard OSEK OS hook routines there are additional hook routine in OSEKturbo OS/MPC5xx used for:

- Performing user's specific operations when no task is running (*IdleLoopHook*)

The OSEKturbo OS provides the following hook routines: *ErrorHook*, *PreTask*Hook, *PostTaskHook*, *StartupHook*, *ShutdownHook* and *IdleLoopHook*[1]. The user must create the code of these routines, the OS only provides description of function prototypes.

[1.] *IdleLoopHook* routine is OSEKturbo extension of OSEK OS.

- *ErrorHook* – this hook is called by the Operating System at the end of a system service which has a return value not equal to *E_OK* (see "Error Interface"). It is called before returning from the service. It is also called when an alarm expires and an error is detected during task activation or event setting.

- *PreTaskHook* – this hook is called before the operating system enters the context of the task. This hook is called from the scheduler when it passes control to the given task. It may be used by the application to trace the sequences and timing of the tasks' execution.

- *PostTaskHook* – This hook is called after the operating system leaves the context of the task. It is called from the scheduler when it switches from the current task to another. It may be used by the application to trace the sequences and timing of tasks' execution.

- *StartupHook* – This hook is called after the operating system startup and internal structures initialization and before initializing System Timer and running scheduler. It may be used by the application to perform initialization actions and task activation.

- *ShutdownHook* – This hook is called when the service *ShutdownOS* has been called. It is called before the Operating System shuts down itself.

- *IdleLoopHook* – This hook is called from scheduler idle loop (see "General"). It is not possible to call any OSEK OS directives from this hook. Hardware dependent code may be placed here. If the user uses *IdleLoopHook* to enter one of lower power modes it is recommended to set *HCLowPower* to FALSE.

Time stamps can be integrated individually into the application software with the help of hook routines *PreTaskHook* and *PostTaskHook*. The user can optionally use the hook routines or establish a watchdog task that takes 'one-shot displays' of the operating system status.

Not all OS services may be called by the hook routines, please refer to the table bellow:

Table 10.1   OSEK OS System Services for Hook Routines

Service	Hook routines				
	Error Hook	**PreTask Hook**	**PostTask Hook**	**Startup Hook**	**Shutdown Hook**
ActivateTask	--	--	--	-	--
TerminateTask	--	--	--	--	--
ChainTask	--	--	--	--	--
Schedule	--	--	--	--	--
GetTaskId	allowed[a]	allowed	allowed	--	--
GetTaskState	allowed	allowed	allowed	--	--
DisableAllInterrupts	--	--	--	--	--
EnableAllInterrupts	--	--	--	--	--
SuspendAllInterrupts	allowed	allowed	allowed	--	--
ResumeAllInterrupts	allowed	allowed	allowed	--	--
SuspendOSInterrupts	--	--	--	--	--
ResumeOSInterrupts	--	--	--	--	--
GetResource	--	--	--	--	--
ReleaseResource	--	--	--	--	--
SetEvent	--	--	--	--	--
ClearEvent	--	--	--	--	--
GetEvent	allowed	allowed	allowed	--	--
WaitEvent	--	--	--	--	--
InitCounter	--	--	--	--	--
CounterTrigger	--	--	--	--	--
GetCounterValue	allowed	allowed	allowed	--	--

Table 10.1   OSEK OS System Services for Hook Routines

Service	Hook routines				
	Error Hook	**PreTask Hook**	**PostTask Hook**	**Startup Hook**	**Shutdown Hook**
GetCounterInfo	allowed	allowed	allowed	--	--
GetAlarmBase	allowed	allowed	allowed	--	--
GetAlarm	allowed	allowed	allowed	--	--
SetRelAlarm	--	--	--	--	--
SetAbsAlarm	--	--	--	--	--
CancelAlarm	--	--	--	--	--
SendMessage	allowed for unqueued messages	--	--	--	--
ReceiveMessage	allowed for unqueued messages	--	--	--	--
GetActiviveApplicationMode	allowed	allowed	allowed	allowed	allowed
StartOS	--	--	--	--	--
ShutdownOS	allowed	--	--	allowed	--
GetRunningStackUsage	allowed	allowed	allowed	--	--
GetStackUsage	allowed	allowed	allowed	--	--
GetTimeStamp	allowed	allowed	allowed	--	--

a. It may happen that currently no task is *running*. In this case the service returns the task Id *INVALID_TASK*.

NOTE	It is not possible to call any OSEK OS services from *IdleLoopHook* hook routine.

Error Handling

Error Interface

The hook routine *ErrorHook* is provided to handle possible errors detected by the OSEK Operating System. Its basic framework is predefined and must be completed by the user. This gives the user a choice of efficient centralized or decentralized error handling.

The special system routine *ShutdownOS* is intended to shut down the system in case of the fatal error. *ShutdownOS* may be called by the user on experiencing a fatal error. These service routine is provided by the OSEK Operating System as opposed to the *ErrorHook* routine, which should be written by the user. User hook *ShutdownHook* is called by *ShutdownOS.*

The OSEK OS *ErrorHook* is called with a parameter that specifies the error. It is up to the user to decide what to do, depending on which error has occurred. According to OSEK OS specification, if system service is called from *ErrorHook* user's hook and this service does not return E_OK error code, then *ErrorHook* is not called. Therefore nested *ErrorHook* calls are blocked by OSEK OS.

The OSEKturbo OS specifies the following error codes:

Table 10.2   OSEK OS Error Codes

Name	Value	Type
E_OK	0	No error, successful completion
E_OS_ACCESS	1	Access to the service/object denied
E_OS_CALLEVEL	2	Access to the service from the ISR is not permitted
E_OS_ID	3	The object ID is invalid
E_OS_LIMIT	4	The limit of services/objects exceeded
E_OS_NOFUNC	5	The object is not used, the service is rejected
E_OS_RESOURCE	6	The task still occupies the resource
E_OS_STATE	7	The state of the object is not correct for the required service
E_OS_VALUE	8	A value outside of the admissible limit

Table 10.2   OSEK OS Error Codes

Name	Value	Type
E_OS_SYS_STACK[a]	17	Task stack overflow
E_OS_SYS_ORDER [a]	18	Incorrect order of function calling
E_OS_SYS_MAINSTACK[a]	19	Main stack overflow
E_OS_SYS_ISRSTACK[a]	20	ISR stack overflow
E_COM_BUSY	33	Message is in use by application task/function
E_COM_ID	35	Invalid message name passed as parameter
E_COM_LIMIT	36	Overflow of FIFO associated with queued messages
E_COM_LOCKED	39	Rejected service call, message object locked due to a pending operation
E_COM_NOMSG	41	No message available

[a]. E_OS_SYS_xxx are not defined in the OSEK OS v.2.2 specification. This is OSEKturbo extension of the OSEK OS.

Errors committed by the user in direct conjunction with the Operating System can be intercepted to a large extent via the Extended Status of the Operating System, and displayed. This results in an extended plausibility check on calling OS services.

Macros for ErrorHook

The special macros are provided by OS to access the ID of the service that caused an error and it's first parameter (only if it is an object ID) inside *ErrorHook* routine:

• the macro *OSErrorGetServiceId*() returns the service identifier where the error has been risen. The service identifier is of type *OSServiceIdType.* Possible values are *OSServiceId_xxxx,* where xxxx is the name of the system service; if the error occured not inside a service function, but in the OS dispatcher then the special value *OSServiceId_NoService*[1] is returned.

• the macros of type <*OSError_serviceID_parameterID*()>, where serviceID is the name of the service and parameterID is the name of the first parameter, returns the value of the first parameter.

[1.] it is OSEKturbo extension of OSEK OS, not specified in OSEK OS v.2.2 specification

For example, *OSError_ActivateTask_TaskID*() returns the task identifier if *ErrorHook* was called because of error detected in ActivateTask.

To work with the *OSErrorGetServiceId* macro, the user has to set the *USEGETSERVICEID* attribute to TRUE. To work with macros for parameter access, the *USEPARAMETERACCESS* attribute shall be set to TRUE. The list of service identifiers and macros for parameter access are provided in *Quick Reference* section of the *OSEKturbo OS/MPC5xx v.2.2.1, User's Manual.*

Extended Status

The OSEK Operating System version with `Extended Status` requires more execution time and memory space than the release version due to the additional plausibility checks it offers. However, many errors can be found in a test phase. After they have all been eliminated, the system can be recompiled with `Standard Status` to the release version.

The following example can illustrate Extended Status usage:

- If a task is activated in the release version, only 'OK' is returned. In the Extended Status version, the additional status like 'Task not defined', 'Task already activated' can be returned. These extended messages must no longer occur in the target application at the time of execution, i.e., the corresponding errors are not intercepted in the operating system's release version.

Possible Error Reasons

Errors in the application software are typically caused by:

- Errors on handling the operating system, i.e. incorrect configuration / initialization / dimensioning of the operating system or non-observance of restrictions regarding the OS service.

- Error in software design, i.e. unwise choice of task priorities, generation of deadlocks, unprotected critical sections, incorrect dimensioning of time, inefficient conceptual design of task organization, etc.

Start-up Routine

The special system routine *StartOS* is implemented in the OSEK Operating System to allocate and initialize all dynamic system and application resources in RAM. This routine is called from the main() function of the application with the application mode as parameter (see "Application Modes") and pass the control to the scheduler to schedule the first task to be running. User hook *StartupHook* is called after operating system startup and before the system and second timers initialization and running scheduler. See "Sample Application" for details.

The figure below shows system startup.

Figure 10.1   System Startup in the OSEK OS
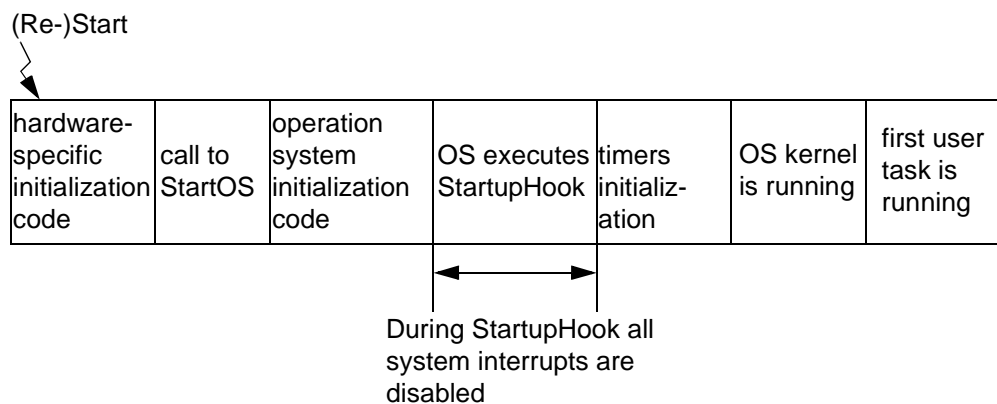
Initializing system and second timers after *StartupHook* avoids loss of timer interrupts during *StartupHook* execution.

Application Modes

Application modes are intended to allow OSEK OS to come under different modes of operation. The application modes differ in the set of autostarted Tasks and Alarms. Once the operating system has been started, it is not possible to change the application mode.

OSEKturbo OS supports up to 8 application modes.

System Shutdown

The special *ShutdownOS* service exists in OSEK OS to shut down the operating system. This service could be requested by the application or requested by the operating system due to a fatal error.

When *ShutdownOS* service is called with a defined error code, the operating system will shut down and call the hook routine *ShutdownHook.* The user is free to define any system behavior in *ShutdownHook* e.g. not to return from the routine. If *ShutdownHook* returns, the operating system enters lower power mode with all interrupts disabled with the lowest Power-down mode.

It is possible to restart OS by calling setjmp() before calling *StartOS*() and then calling longjmp() in *SHUTDOWNHOOK.*

Programming Issues

Configuration Options

The following configuration options affect error handling and hook routines:

- *ERRORHOOK*
  If this option is turned on, the *ErrorHook* is called by the system for error handling

- USEGETSERVICEID

  If this option is turned on, it allows to get the service ID inside ErrorHook.

- USEPARAMETERACCESS

  If this option is turned on, it allows to get the first service parameter value inside ErrorHook.

- *PRETASKHOOK*
  If this option is turned on, the *PreTaskHook* is called by the system before context switching

- *POSTTASKHOOK*
  If this option is turned on, the *PostTaskHook* is called by the system before context switching

- *STARTUPHOOK*
  If this option is turned on, the *StartupHook* is called by the system at the end of the system initialization

- *SHUTDOWNHOOK*
  If this option is turned on, the *ShutdownHook* is called by the system when the OS service *ShutdownOS* has been called

- *IdleLoopHook*
  If this OSEKturbo specific option is turned on, the *IdleLoopHook* is called from the scheduler idle loop

11

System Configuration

This chapter describes possible OSEK OS versions, configuration options and the configuration mechanism.

This chapter consists of the following sections:

General

The OSEK Operating System is fully statically configured one. All system properties, the number of system objects and their parameters (characteristics of tasks, counters, alarms, messages, etc.), run time behavior are defined by the user. Such approach allows the user to create various range of applications with exactly defined characteristics. Different memory and performance requirements can be easily satisfied with such modular approach.

All application parameters are defined in the special configuration file. This file must conform OIL grammar rules. It is processed by the separate *System Generator utility (SG)*[1]. The System Generator analyzes statements in the configuration file and builds output C-language files needed to compile OS source files and to compile and link an application with the specified features. During its execution SG reports to the user about the errors. The System Generator produces header and source code files that defines all properties and objects in terms of the C language. These files are to be compiled and linked together with the user's and OS source code.

[1.] One version of SG is delivered - the 32-bit version ('sysgen.exe') for Windows 98 and Windows 2000.

Application Configuration File

Application configuration file contains the statements which define the system properties and objects. Such file can have any extension and the extension '.oil' is suggested by default. The file of this format is processed by the SG utility.

As a result of application configuration file processing SG produces three types of standard C-language files as it is described in "Application Configuration" and optional ORTI file as it described in "Using OS Extended Status for Debugging". SG produces two header files and one source file. These files provides the code for all system tables, descriptors, arrays etc. both in ROM and RAM according to the user specified application configuration.

OIL Concept

OSEK Implementation Language (OIL) is the specially designed language for development of embedded applications based on OSEK concept. OIL is used to describe the application structure (application configuration) as a set of system objects with defined links. OIL allows the user to write an application configuration as a text file. These files have predefined structure and special (standard) grammar rules.

All system objects specified by OSEK and relationships between them can be described using OIL. OIL defines standard types for system objects. Each object is described by a set of attributes and references.

All keywords, attributes, object names, and other identifiers are case-sensitive.

OIL File

The OIL file contains two parts – one for the definition of implementation specific features (Implementation Definition) and another one for the definition of the structure of the application located on the particular CPU (Application Definition).

In the very beginning of an OIL file the number of the version of OIL is indicated. The keyword OIL_VERSION is used for this purpose. For example:

```
OIL_VERSION = "2.3";
```

OIL Format

The Standard OIL is intended to configure OSEK OS Operating System (OS). It strictly defined by the *OSEK/VDX System Generation OIL: OSEK Implementation Language, v.2.3, 10 September 2001* specification.

Implementation Definition

The Implementation Definition defines implementation specific features for the particular OSEK implementation for which this application is developed.

The implementation can limit the given set of values for object attributes (e.g. restrict the possible OS conformance classes).

It is not allowed to exclude any standard attributes from the particular OSEK implementation. Additional non-standard attributes can be defined for the objects for the particular OSEK implementation.

The include mechanism (see "Include Directive") can be used to define the implementation definition as a separate file. Thus corresponding implementation definition files are developed and delivered with particular OSEK implementations and then included in user's OIL files. The OSEKturbo OS/MPC5xx implementation is described in the "os22mpc.oil" file which are delivered in the package.

Implementation Definition Grammar

Implementation Definition part starts with keyword *IMPLEMENTATION* and implementation name.

The structure for Implementation Definition part is shown using the following syntax:

```
IMPLEMENTATION <name> {
    <object_descriptions>
};
```

All objects within Implementation Definition part are described using the same syntax.

```
<object_type> {
    <property_definitions>
};
```

Object type is defined by the object keyword. For OSEKturbo OS/ MPC5xx implementation the following object types are implemented:

```
OS, APPMODE, TASK, ISR, RESOURCE, EVENT, COUNTER, ALARM, MESSAGE,
COM, NM
```

The set of object properties are to be defined within the object description. Both implementation specific attribute and reference shall be defined before it is used.

The attribute definition has the following structure:

```
<attr_type> [ WITH_AUTO ] [ <attr_range> ]
<attr_name> [ = <default_value> ] [
<multiple_specifier> ];
```

The attribute type and attribute value range (if it exists) shall be defined. The range of attribute values can be defined in two ways: either the minimum and maximum allowed attribute values are defined (the [0..12] style) or the list of possible attribute values are presented (like C enumeration type).

The WITH_AUTO specifier can be combined with any type. If WITH_AUTO can be specified it means that this attribute can have the value AUTO and the possibility of automatic assignment.

Data types defined for OIL are listed below. Note that these data types are not necessarily the same as the corresponding C data types.

- UINT32 - any unsigned integer number (possibly restricted to a range of numbers. This data type allows to express any 32 bit value in the range of $[0..(2^{32}-1)]$.

- INT32 - any signed integer number in the range of $[-2^{31}..(2^{31}-1)]$.

- UINT64 - any unsigned integer number in the range $[0..(2^{64}-1)]$.

- INT64 - any signed integer number in the range $[-2^{63}..(2^{63}-1)]$.

- FLOAT - any floating point number according to IEEE-754 standard (Range: +/- 1,176E-38 to +/-3,402E+38).

- ENUM – the list of possible values shall be presented. Any value from the list can be assigned to an attribute of this type. ENUM types can be parameterized, i.e. the particular enumerators can have parameters. The parameter specification is denoted in curly braces after the enumerator. Any kind of attribute type is allowed as parameter of an enumerator.

- BOOLEAN – The attribute of this type can have either TRUE or FALSE value. BOOLEAN types can be parameterized, i.e. the particular boolean values can have parameters. Parameter specification are denoted in curly braces after an explicit enumeration of the boolean values. Any kind of attribute type is allowed as parameter of a boolean value.

- STRING – Any 8-bit character sequence enclosed in double-quotes, but not containing double-quotes can be assigned to this attribute.

A reference is a special type of value intended to define links between system objects. The reference definition has the following structure:

```
<object_type> <reference_name> [
<multiple_specifier> ];
```

The reference type is taken from the referenced object (e.g. a reference to a task shall use the TASK_TYPE keyword as reference type). A reference can 'point to' any system object.

Multiple reference is the possibility to refer to several objects of the same type with one OIL statement. For example the task can refer to several events. If the reference shall be defined as a 'multiple' reference then the '[]' brackets shall be present after the reference name.

An attribute can have a subattributes which are described in curly brackets.

Application Definition

In the application definition the OSEK application is composed from a set of system objects. In general the application can contain more than one system object of the same type.

Since an application is performed on CPU the entity called *CPU* is introduced as the top of the description. This entity encompasses all local objects such as tasks, messages, etc. Therefore, *CPU* can be considered as a container for application objects. This concept is introduced to provide future OIL evolution towards to distributed system support. This entity is identified by the keyword CPU.

Object Definition

All objects are described using the same syntax.

```
<object_type> <object_name> {
    <property_definitions>
};
```

Objects are labeled by keywords which shall be written in upper case. Object attributes and references are also labeled by the keywords. The keywords are introduced in "System Objects Definition". After an object keyword the object name must follow. Name is combined from any symbols up to 32 symbols long.

A set of attributes and references belonging to an object is enclosed in curly brackets, like in C language.

All assignments are made via the '=' operator. Each statement ends with semicolon - ';' like in the C language. A reference is represented as a reference type keyword assigned with a name of the object referenced. If multiple reference pointed to the set of objects several references shall be used. Here is example for task referencing to own events:

```
EVENT = MyEvent1;
EVENT = MyEvent2;
```

Include Directive

The preprocessing directive to include other OIL files is allowed in any place of the OIL file. This statement has the same syntax as in ANSI-C:

```
#include <filename.oil>
#include "[path]filename.oil"
```

The file name can be optionally preceded by a directory specification. The quoted form means that a header file is being looked for in the current directory first, then along the path specified by the "/I" command-line option, then along paths specified by the special environment variable. The angle-bracket form means that a header file is being looked for first along the path specified by the "/I" command-line option, then along paths specified by the special environment variable.

Comments

An OIL file may contain comments. The '/*' and '//' characters define the start of a comment. Any characters after '//' are considered as a comment and the end of line (EOL) terminates the comment. Any characters after
'/*' are considered as comments and the end of the comment is defined by '*/'. Nested comments are not allowed in OIL.

File Structure

Any file in the Standard OIL format describes an application for a single CPU and, in general, must have the following structure:

```
OIL_VERSION = <version>;
IMPLEMENTATION <name> {// Implementation definition
  <OBJECT_TYPE> {
    ...list of implementation specific object attributes...
  };
  ...
};

CPU <name> {             // Definition of the application on CPU
  <OBJECT_TYPE> <object_name>
  {                      // System object definition
```

```
    <ATTRIBUTE> = <value>;
    <REFERENCE> = <object_name>;
    ... list of object attributes and references ...
  };
  ... list of objects ...
};
```

Configuration File Rules

The application configuration files must conform some simple rules to be successfully processed. The rules are:

- All objects are described using the OIL syntax.

- Each object must have a unique name. Each object may be divided into several parts.

- All object names are accessible from the application.

- An attribute defines some object properties (for example, the task priority). Attributes that take a single value may only be specified once per object. Attributes that take a list of values must be specified as multiple statements.

- An object can have a set of references to other system objects. Per object there may be more than one reference to the same type of object.

- Values must be defined for all standard attributes of all objects, except for multiple attributes which can be empty.

- All statements must be written without errors;

- It is recommended to avoid conflicting statements (e.g., the STACKSIZE is defined and no EVENT(s) is defined for a task) since it leads to error or warning messages.

For More Information: www.freescale.com

12

System Objects Definition

This chapter describes objects that are controlled by the Operating System - tasks, resources, alarms, messages, counters, ISRs and even the OS itself - are considered as system objects.

This chapter consists of the following sections:

- General
- OS Definition
- Task Definition
- ISR Definition
- Resource Definition
- Event Definition
- Counter Definition
- Alarm Definition
- Message Definition
- Application Modes Definition
- COM Definition
- NM Definition
- OSEKturbo Performance Dependency

General

All objects that are controlled by the Operating System – tasks, resources, alarms, messages, counters, ISRs and even the OS itself - are considered as system objects. Each of them has its unique characteristics defined by the user. To specify parameters for each system object the special statements are used for each object. All statements are described below in detail.

Each group of attributes has scheme which described attributes nesting. Possible attribute values are placed in angle brackets and

separated by slash. Default value is marked as bold text. If default value is present, the attribute can be omitted. Scheme includes all attributes, but some of them can be used if parent attribute has the determined value only. You can find description of these dependencies below the scheme.

OS Definition

The *Operating System* is the mandatory object for any application. This object is used to define OS configuration parameters. The keyword OS is used for this object type. Only one OS object can be defined in the application. See "Operating System Architecture" for more detailed information about OS. The syntax of the OS object definition is as follows:

```
OS <name>{
      <attributes>
    };
```

OS object's attributes can be divided into groups which correspond to appropriate system objects and their interaction. Nested structure of OS object definition are displayed on the syntax schemes of each attribute group.

Different groups of related attributes are described below. Brief explanations are provided. All these attributes should be defined inside the scope of the OS object.

Global System Attributes

This group of attributes represents system features which are common for the whole system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
STATUS = <STANDARD / EXTENDED>;
CC = <BCC1 / ECC1 / AUTO>;
DEBUG_LEVEL = <0 / 1 / 2 / 4>;
BuildNumber = <TRUE / FALSE>;
FastTerminate = <TRUE / FALSE>;
FastScheduler = <TRUE / FALSE>;
```

```
MessageCopyAllocation = <USER /OS>;
ResourceScheduler = <TRUE / FALSE>;
```

- *STATUS* (ENUM)
  The attribute specifies the debugging ability of OS, defines whether additional run-time error checks are supported by OS for OSEK API calls or not. If the system has the *extended* status some additional checks are performed to detect run-time errors. The extended status adds approximately 20% of code, and increases timing accordingly. This mode is considered to be a debugging mode. The standard status of OS performs only very limited set of checks, the performance is increased and the amount of consumed memory is decreased.

  As a general approach, it is recommended to start application development with extended status to discover configuration and run-time problems. For debugged applications the status may be set to standard to reduce code size and advance timing.

- *CC* (ENUM or AUTO)
  The attribute specifies the conformance class which is supported by the OS. However all features of the OS may be selected by means of using other OS additional properties. Therefore, even for given conformance class the functionality may be reduced or increased according to user's needs.

  If the value is *AUTO* then conformance class is defined according to TASKs definitions.

- *DEBUG_LEVEL* (ENUM)
  This OSEKturbo specific attribute specifies the ORTI support in OS. If the system has the *DEBUG_LEVEL* = 0, ORTI is not supported. If the attribute is set to 1 or higher, the static ORTI mode is turned on. If the attribute is set to 2, static and run time ORTI (running task and system calls identification) modes will be used. If the attribute is set to 4, then dynamic and static ORTI and debugging services are implemented. See "Debugging Application" for details.

- *BuildNumber* (BOOLEAN)
  This OSEKturbo specific attribute specifies whether build number in ASCII form should be incorporated into OS binary image (ROM code) or not. If *BuildNumber* is set to FALSE then the user shall remove "OsBuildNumber" entry from linker command file (which is generated from makefile) for CodeWarrior compiler.

- *FastTerminate* (BOOLEAN)
  This OSEKturbo specific attribute specifies whether the fast versions of *Terminate/ChainTask* is used in BCC1 class.

- *FastScheduler* (BOOLEAN)
  This OSEKturbo specific attribute specifies whether the OS places global variabnles, used by scheduler, into the CPU general purpose registers.

- *MessageCopyAllocation* (ENUM)
  This OSEKturbo specific attribute specifies whether System Generator generates copies of messages in global memory or message copies are allocated by the user. The user can place copies of messages either in global memory or in local memory. This attribute affects all message copies in the application.

- *ResourceScheduler* (BOOLEAN)
  This OSEKturbo specific attribute specifies whether *RES_SCHEDULER* should be supported or not.

CPU Related Attributes

This group of attributes provides with possibility to tune the selected hardware. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
TargetMCU = <MPC / MPC555 / MPC565 / MPC563 /
MPC561 /> {
    ClockFrequency = <integer /4000>;
    ClockDivider = 1;
    ClockMultiplier = <integer / 1>;
    SysTimer = <HWCOUNTER / SWCOUNTER / NONE> {
      COUNTER = <name of COUNTER>;
      ISRPRIORITY = <integer>;
      Period = <integer / AUTO>;
      TimerHardware = <name of hardware timer> {
        Prescaler = <USER / OS> {
          Value = <integer / AUTO>;
        };
        TimerModuloValue = <integer / AUTO>;
        Freeze = <TRUE / FALSE>;
        RTSEL = <EXTCLK / OSCM>;
      };
    };
```

```
      SecondTimer = <HWCOUNTER / SWCOUNTER / NONE> {
        COUNTER = <name of COUNTER>;
        ISRPRIORITY = <integer>;
        Period = <integer / AUTO>;
        TimerHardware = <name of hardware timer> {
          Prescaler = <USER / OS> {
            Value = <integer / AUTO>;
          };
          TimerModuloValue = <integer / AUTO>;
          Freeze = <TRUE / FALSE>;
          RTSEL = <EXTCLK / OSCM>;
        };
      };
      HCLowPower = <TRUE / FALSE>;
  };
```

- *TargetMCU* (ENUM)
  This OSEKturbo specific attribute defines the target for which the OS will be configured.

  It is highly recommended to use sample makefile as a basis for building application to provide consistency in compiler/linker settings.

  Value MPC corresponds to any MCU of this MCU family. Any target specific parts of code (such as timer support) are disabled in this case.

- *ClockFrequency* (UINT32)
  This OSEKturbo specific attribute specifies timers input frequency (OSCM or EXTCLK, depending on user setting of *RTSEL* for PIT and RTC; OSCM for DEC and TB timers)in kHz for calculating prescaler value and timer modulo value. This attribute shall be defined if any of *Period, Prescaler/Value* or/and any *TimerModuloValue* is *AUTO*. Otherwise this attribute is ignored.

- *ClockDivider* (UINT32)
  This OSEKturbo specific attribute specifies PLL divider for calculating input timer frequency. The attribute is applicable if the *TimerHardware* attribute has DEC, TB0 or TB1 value. It is the user responsibility to initialize hardware to appropriate values. Value of the attribute shall be equal to actual divider, not the value written into the HW register.

- ***ClockMultiplier*** (UINT32)
  This OSEKturbo specific attribute specifies PLL multiplier for calculating input timer frequency. The attribute is applicable if the *TimerHardware* attribute has DEC, TB0 or TB1 value. It is the user responsibility to initialize PLL hardware with appropriate values. Value of the attribute shall be equal to actual multiplier, not the value written into the HW register.

- ***SysTimer*** (ENUM)
  This OSEKturbo specific attribute specifies whether the internal OS system timer is used or not. If *SysTimer* is set to SWCOUNTER or HWCOUNTER, interrupt services are turned on automatically. The attribute can not be defined, if *TargetMCU* is set to *MPC*.

  If this attribute is SWCOUNTER or HWCOUNTER, then specific subattributes can be defined for the system timer.

- ***SecondTimer*** (ENUM)
  This OSEKturbo specific attribute specifies whether the internal OS second timer is used or not. The attribute can not be defined, if *TargetMCU* is set to *MPC*.

  If this attribute is SWCOUNTER or HWCOUNTER, then specific subattributes can be defined for the second timer. The *SecondTimer* attribute and its subattributes can be defined only if *SysTimer* value is equal to SWCOUNTER or HWCOUNTER. The *SecondTimer* attribute can be defined to HWCOUNTER only if *SysTimer* value is set to HWCOUNTER.

- ***COUNTER*** (reference)
  The reference specifies the *COUNTER* which shall be attached to the system or second timer. This attribute shall be defined for system timer and for second timer if respectively *SysTimer* value and *SecondTimer* value is SWCOUNTER or HWCOUNTER. The same counter can not be attached to both System and Second timers.

- ***ISRPRIORITY*** (UINT32)
  The attribute specifies priority of system timer (second timer) interrupt handler. This attribute shall be defined inside the scope of the *SysTimer* and *SecondTimer* attributes if these attributes are set to SWCOUNTER or HWCOUNTER. The rules defined for *PRIORITY* attribute of ISR object are also applicable for this parameter..

The values of ISRPRIORITY attribute for the different timer hardware are presented below. The value depends on the *InterruptDispatcher* value - Enhanced or not.

Timer hardware	MPC555	MPC565/563/561	
		not Enhanced	Enhanced
DEC	16	16	48
PIT,RTC, TB0,TB1	0, 2, 4, 6, 8, 10, 12, 14	0, 2, 4, 6, 8, 10, 12, 14	4, 10, 16, 22, 28, 34, 40, 46

When both timers are HWCOUNTER and *TimerHardware* are equal to TB0 and TB1 this attribute shall be equal for both timers because it configures the same interrupt service routine.

- *Period* (UINT32)
This OSEKturbo specific attribute specifies period of a tick of the system (second) counter in nanoseconds. This attribute can be defined inside the scope of the *SysTimer* and *SecondTimer* attributes if these attributes are set SWCOUNTER or HWCOUNTER.

The *Period* attribute shall be defined if the corresponded *Prescaler/Valu*e or */* and *TimerModuloValue* is AUTO. This attribute is ignored if corresponded *Prescaler/Valu*e and *TimerModuloValue* are not AUTO.

NOTE *OSTICKDURATION* and *OSTICKDURATION2* constants are calculated from the *SysTimer/Period* value or *SecondTimer/Period* value respectively if any of corresponding *Prescaler/Value* and *TimerModuloValue* is AUTO. Otherwise *OSTICKDURATION* and *OSTICKDURATION2* are calculated from the values of the corresponding *Prescaler/Value* and *TimerModuloValue* attributes.

- *TimerHardware* (ENUM)
This OSEKturbo specific attribute is intended to select the hardware interrupt source for the system counter or the second counters (among the accessible MCU devices). This attribute inside the *SysTimer* and *SecondTimer* attributes can be defined if the values of these attributes are HWCOUNTER or SWCOUNTER only. The possible values for SWCOUNTER are: PIT, DEC, RTC for MPC5xx; the possible values for HWCOUNTER are TB0, TB1 for MPC5xx.
The *TimerHardware* attributes in *SysTimer* and *SecondTimer* blocks can not have the same value.

See "PowerPC Platform-Specific Features" for details about possible meanings of these parameters.

- *Prescaler* (ENUM)
  This OSEKturbo specific attribute specifies whether prescaler value shall be initialized during OS startup or prescaler will be set by user application. This attribute shall be defined if the *SysTimer* (*SecondTimer*) value is SWCOUNTER or HWCOUNTER only.

  The *Prescaler* attribute in the *SysTimer* attribute scope and the *Prescaler* attribute inside the *SecondTimer* attribute shall have the same value.

- *Value* (UINT32 or AUTO) – inside *Prescaler* attribute
  This OSEKturbo specific attribute specifies initial prescaler value for the selected system or second timer hardware. The value of this attribute is fully hardware-dependent. For more details see "PowerPC Platform-Specific Features". Note that the *Prescaler/ Value* attribute value is not equal to divide factor of timer hardware.

  This attribute can be *AUTO* only if *Prescaler* value is *OS*. In case of *AUTO*, prescaler value is calculated during system generation. This attribute shall be defined if *Prescaler* value is USER. If the *TimerModuloValue* attribute is not *AUTO*, the *Value* shall be defined by numeric value. In case of *AUTO* prescaler value is calculated from the values of *ClockFrequency*, *ClockDivider*, *ClockMultiplier* and corresponded *Period* attributes during system generation.

  The *Value* attribute in the *SysTimer* attribute scope and the *Value* attribute inside the *SecondTimer* attribute shall have the same value if values define the same hardware parameter, or any of them shall be AUTO.

- *TimerModuloValue* (UINT32)
  This OSEKturbo specific attribute specifies timer modulo value. The value of this attribute is fully hardware-dependent. For more details see "PowerPC Platform-Specific Features".

  This attribute can be defined if *SysTimer* or *SecondTimer* value is *SWCOUNTER* only. If the attribute is set *AUTO*, timer modulo value is calculated from the values of the *ClockFrequency*, *ClockDivider*, *ClockMultiplier* and corresponded *Period* attributes during system generation.

- *Freeze* (BOOLEAN)
  This attribute specifies timer freeze bit value. If this attribute has

value TRUE, timer will be frozen during interrupt processing in the debug mode.

- *RTSEL* (ENUM)
This attribute specifies clock source for RTC and PIT timers. The EXTCLK value means that external clock will be used as clock source, the OSCM value means that the main crystal oscillator will be used as clock source. This attribute shall have same value for System and Second timers if they are PIT and RTC.

- *HCLowPower* (BOOLEAN)
This OSEKturbo specific attribute defines that low power mode shall be used when there is no ready or running tasks and scheduler goes into idle loop.

  In general, it is recommended to define this attribute to reduce power consumption.

Stack Related Attributes

These attributes define stack support in the system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
IsrStackSize = <integer>;
StackOverflowCheck = <TRUE / FALSE>;
```

- *IsrStackSize* (UINT32)
This OSEKturbo specific attribute specifies the ISR stack size in bytes. The stack is used in ISRs category 2. In this case the current status of the processor is saved onto the current stack, and stack is switched to the interrupt stack. This stack should be big enough to hold all nested interrupt stack frames in the system.

  This attribute shall be defined if any ISR category 2 (include system or second timer) are defined and Extended tasks are present in the configuration. Otherwise single stack is used for Basic tasks and ISRs and this attribute is ignored.

- *StackOverflowCheck* (BOOLEAN)
This OSEKturbo specific attribute turns on stack overflow runtime checking and stack usage services. If this attribute is switched on, *ErrorHook* is defined and OS detects stack overflow, then the *ErrorHook* is called with appropriate error status code. This mode is intended for debugging.

Task Related Attributes

This group of attributes controls task features. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
TimeScale = <TRUE / FALSE> {
     ScalePeriod = <integer / AUTO>;
     TimeUnit = <ticks / ns / us /ms>;
     Step = <SET> {
       StepNumber = <integer>;
       StepTime = <integer>;
       TASK = <name of TASK>;
     };
};
```

- *TimeScale* (BOOLEAN)
  This OSEKturbo specific attribute specifies Time Scale mechanism.

- *ScalePeriod* (UINT32)
  This OSEKturbo specific attribute specifies full period of time scale in chosen measurement units. It is used during system generation time to check that *ScalePeriod* is equal to sum of *StepTime* attributes of all steps. The attribute is ignored if it is AUTO.

- *TimeUnit* (ENUM)
  This OSEKturbo specific attribute specifies measurement units: ticks means ticks of System Timer, ns means nanoseconds, us - microseconds, and ms - milliseconds.

- *Step* (ENUM)
  This OSEKturbo specific attribute defines one of step elements in the Time Scale.

- *StepNumber* (UINT32)
  This OSEKturbo specific attribute specifies the order of steps. The numbers shall be unique.

- *StepTime* (UINT32)
  This OSEKturbo specific attribute specifies time to next task activation in measurement units chosen by means of the *TimeUnit* attribute.

- *TASK* (reference)
  The reference specifies the task to be activated.

Interrupt Related Properties

This group of attributes defines parameters of ISR execution. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
UnorderedExceptions = <TRUE / FALSE>;
InterruptDispatcher = <None / OneLevel /
                       MultiLevel / Enhanced>;
```

- *UnorderedExceptions* (BOOLEAN)
  The attribute specifies either unordered exception handling is supported or not.
  To be able to recover from PowerPC unordered exceptions it is necessary to manipulate with MSR RI bit during exception prologue/epilogue and during task context switch. If this attribute is *TRUE*, then OS will set MSR RI bit after saving SRR0 and SRR1 registers and will clear MSR RI bit before loading SRR0 and SRR1 registers during ISR prologue/epilogue and task context switch.

  OSEKturbo OS/MPC5xx itself does not use RI bit, but it may be necessary to set *UnorderedExceptions* to TRUE for some debuggers to work properly.

- *InterruptDispatcher* (ENUM)

  The attribute specifies the way of the external interrupts processing. It can have one of the following values:

  – None - the interrupt dispatcher is not supported. In this case user have to care about the external interrupt processing;

  – OneLevel - the interrupt dispatcher is supported, but multilevel external interrupt emulation is not supported. In this case all external interrupts will be disabled during the interrupt processing.

  – MultiLevel - the interrupt dispatcher and multilevel external interrupt emulation are supported. In this case interrupt processing can be interrupted by the other interrupt of the higher level. The level of ISR is specified by its *PRIORITY*;

  – Enhanced - the interrupt dispatcher is supported for Enhanced Interrupt Controller for MPC565/563/561. The level of ISR is specified by its *PRIORITY*.

Hook Routines Related Attributes

These attributes define hook routines support in the system. The attributes should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value are shown in bold type):

```
STARTUPHOOK = <TRUE / FALSE>;
SHUTDOWNHOOK = <TRUE / FALSE>;
PRETASKHOOK = <TRUE / FALSE>;
POSTTASKHOOK = <TRUE / FALSE>;
ERRORHOOK = <TRUE / FALSE>;
USEGETSERVICEID = <TRUE / FALSE>;
USEPARAMETERACCESS = <TRUE / FALSE>;
IdleLoopHook = <TRUE / FALSE>;
```

- *STARTUPHOOK* (BOOLEAN)
  The attribute defines whether the user's-provided hook is called by the system after startup but before starting dispatcher and initializing system timer or not (the *StartupHook* hook routine). This hook may be used by the application to perform hardware initialization actions, etc.

  The alternative way is to make such initialization steps in the task, which starts automatically. The hook is called with disabled interrupts.

- *SHUTDOWNHOOK* (BOOLEAN)
  The attribute defines whether the user's-provided hook is called by the system during system shutdown or not (the *ShutdownHook* hook routine). The main purpose of this hook is to shutdown initialized hardware devices like timers, network controllers, etc. Besides, the reason for shutdown may be obtained through the error code.

  This hook is called after system timer shutdown (if system timer is configured in the system). Interrupts are disabled in this hook .

- *PRETASKHOOK* (BOOLEAN)
  The attribute defines whether the user's-provided hook is called from the scheduler code before the operating system enters context of the task or not (the *PreTaskHook* hook routine). In general, this hook is designed for debugging applications by means of tracing current task..

It is not recommended to use this hook in normal working applications, because it adds significant timing overhead. If the attribute is defined, this hook is called for each task, i.e. it is not allowed to use this hook for particular task(s) only.

- **POSTTASKHOOK** (BOOLEAN)
  The attribute defines whether the user's-provided hook is called from the scheduler code after the operating system leaves context of the task or not (the *PostTaskHook* hook routine). In general, this hook is designed for debugging applications by means of tracing current task.

  It is not recommended to use this hook in normal working applications, because it adds significant timing overhead. If the attribute is defined, this hook is called for each task, i.e. it is not allowed to use this hook for particular task(s) only.

- **ERRORHOOK** (BOOLEAN)
  The attribute defines whether the user's-provided hook is called by the system at the end of each system service which returns status not equal to E_OK or not (the *ErrorHook* hook routine). This hook is designed for debugging applications by means of tracing error code, returned by the system service instead of checking error code after each call of system service. This hook increases the OS code with extended error status by approximately 10% (for *STANDARD* status configuration), and increase the timing in case of error during the service call.

  There is no need to check the error status of the each OS service call if this hook is used. This hook is useful as a temporary feature of a working (debugged) applications when some troubles occur. If the attribute is defined, this hook is called from the system service in which error occurs, i.e. it is not allowed to use this hook for particular service(s) only.

- **USEGETSERVICEID** (BOOLEAN)

  The attribute specifies possibility of usage the access macros to the service ID in the error hook.

- **USEPARAMETERACCESS** (BOOLEAN)

  The attribute specifies possibility of usage the access macros to the context related information in the error hook.

- **IdleLoopHook** (BOOLEAN)
  This OSEKturbo specific attribute defines whether the user's-provided hook is called by system from scheduler idle loop

(when there are no tasks in *ready* or *running* state) or not (the *IdleLoopHook* hook routine).

This hook is intended for manipulation with hardware registers (like COP). It is not possible to call any OSEK OS services from this hook.

Floating Point Related Attributes

OSEKturbo OS does not save/restore floating point registers during system startup/shutdown.

By default, OS does not save/restore floating point registers during context switch. Therefore, if it is necessary to perform floating-point calculations inside OSEKturbo OS application, *FloatingPoint* option should be used.

It is not possible to perform floating-point calculations inside ISR handlers.

The following statement should be used to configure floating-point support in OSEK OS:

The attribute should be defined inside the scope of the OS object in accordance with the following syntax (default attribute value is shown in bold type):

```
FloatingPoint = <TRUE / FALSE>;
```

- *FloatingPoint* (BOOLEAN)
  The *FloatingPoint* attribute turns floating-point support. If the attribute is TRUE the floating-point registers are saved/restored during context switch and FP bit in MSR is set during task's execution. If the attribute is FALSE the floating-point registers are not saved/restored during context switch.

Task Definition

Task object is used in OIL file to define task data. Several statements can be in the configuration file, each statement defines one task.

Attributes of this object define task properties. Links with other system objects are defined via references. The keyword TASK is

used for this object type. See <u>"Task Management"</u> for more detailed information about OSEK tasks. The syntax of the task object is as follows:

```
TASK <name of TASK> {
    PRIORITY = <integer>;
    SCHEDULE = <FULL / NON>;
    AUTOSTART = <TRUE / FALSE>{
      APPMODE = <name of APPMODE>;
    };
    ACTIVATION = <1>;
    STACKSIZE = <integer>;
    RESOURCE = <name of RESOURCE>;
    EVENT = <name of EVENT>;
    ACCESSOR =<SENT / RECEIVED> {
      MESSAGE = <name of MESSAGE>;
      WITHOUTCOPY = <TRUE / FALSE>;
      ACCESSNAME = <string>;
    };
};
```

Attributes

The TASK object has the following attributes:

- ***PRIORITY*** (UINT32)
  The attribute specifies priority of the task. The value of this attribute has to be understood as a relative value; this means the values of the PRIORITY attribute show only the relative ordering of the tasks. OSEK defines the lowest priority as zero (0), the bigger value of the *PRIORITY* attribute corresponds to the higher priority. The value range is from 0 to 0x7FFFFFFF.

- ***SCHEDULE*** (ENUM)
  The attribute specifies the run-time behavior of the task. If the task may be preempted by another one at any point of execution - this task attribute shall have the *FULL* value (preemptable task). If the task can be preempted only at specific points of execution (explicit rescheduling points) the attribute shall have the *NON* value (non-preemptable task).

- ***AUTOSTART*** (BOOLEAN)
  The attribute determines whether the task is activated during the system start-up procedure or not.

- *APPMODE* (reference)

  The attribute defines an application mode in which the task is auto-started. The attribute can be defined if the *AUTOSTART* attribute is set to *TRUE*. It may be ommited if only one *APPMODE* is defined.

- *ACTIVATION* (UINT32)
  The attribute defines the maximum number of queued activation requests for the task. OSEKturbo OS does not support multiple activation, so this value is restricted to 1.

- *STACKSIZE* (UINT32)
  This OSEKturbo specific attribute defines the task stack size in bytes. It is applicable for extended tasks only.

- *RESOURCE* (reference)
  This reference is used to define a resource accessed by the task. If the task accesses a resource at run-time this resource shall be pointed. The resource Ceiling priority is calculated as the highest priority of tasks or ISRs accessing this resource. There can be several *RESOURCE* references. This parameter can be omitted.

- *EVENT* (reference)
  This reference is used to define an event the extended task may react on. The task is considered as extended, if any event is reverenced. Otherwise the task considered as basic.

  There can be several *EVENT* references. These events can be cleared and waited for by the task. All task events shall be pointed to define the event mask in case of auto-assignment (see section "Event Definition").

- *ACCESSOR* (ENUM)
  The attribute is used to define type of usage for the message. The task uses the accessor for definition of references to sent or received messages.

- *MESSAGE* (reference)
  The reference specifies the message to be sent or received by the task. This parameter is a single reference, it has to be defined.

- *WITHOUTCOPY* (BOOLEAN)
  The attribute defines whether a local copy of the message is used or not. This attribute has to be defined.

- *ACCESSNAME* (STRING)
  The attribute defines the reference which can be used by the application to access the message data. This attribute has to be defined. The local copy of the message shall be defined with this

name in application code in case an access to the message is processed using local copy.

ISR Definition

This object presents an Interrupt Service Routine. The keyword ISR is used for this object type. The syntax of the ISR object is as follows:

```
ISR <name of ISR> {
    CATEGORY = <1 / 2>;
    PRIORITY = <integer>;
    RESOURCE = <name of RESOURCE>;
    ACCESSOR =<SENT / RECEIVED> {
      MESSAGE = <name of MESSAGE>;
      ACCESSNAME = <string>;
    };
};
```

The same ISR name shall be used for corresponding ISR object declaration and definition (see "Conventions").

Attributes

This object has the following attributes:

- *CATEGORY* (UINT32)
  The attribute specifies category of the Interrupt Service Routine. (see "ISR Categories" for Interrupt Service Routine Categories details).

- *PRIORITY* (UINT32)
  The attribute specifies priority of the interrupt service routine.

  Allowed values for MPC555 applications are 0-16, if *InterruptDispatcher* value is OneLevel *or* MultiLevel. In this case all ISR objects have to have the different value of the *PRIORITY* attribute. All ISRs which intended for not external interrupt handling have to have *PRIORITY* = 16 for MPC555. For MPC565/563/561 the range depends on the *InterruptDispatcher* value. If the *InterruptDispatcher* attribute is OneLevel or MultiLevel, then the *PRIORITY* value range is from 0 to 16. If the *InterruptDispatcher* attribute is Enhanced, the range is from 0 to 48.

NOTE	The value of this attribute has no significance if the *InterruptDispatcher* attribute has value None.

This attribute value is connected with interrupt controller's interrupt levels.

Regular Interrupt Controller Operation

Each one of the SIU interrupt sources (such as PIT and RTC), as well as the interrupt requests generated by the IMB3 modules can be assigned to any of the Level 0..7. Each of the external pins IRQ[0..7] has its own dedicated assigned priority level.

The accordance of the ISR PRIORITY to the interrupt sources in the interrupt controller for MPC555 is shown in following table:

Table 12.1   ISR Priority and Interrupt Sources for MPC555

ISR PRIORITY	Interrupt sources
15	IRQ0
14	Level 0
13	IRQ1
12	Level 1
11	IRQ2
10	Level 2
9	IRQ3
8	Level 3
7	IRQ4
6	Level 4
5	IRQ5
4	Level 5
3	IRQ6
2	Level 6
1	IRQ7
0	Level 7

For more detailed information about MPC555 interrupt controller see MPC555 User's Manual.

ISRs which have interrupt source beside interrupt controller have to have *PRIORITY* 16 for MPC555.

Enhanced Interrupt Controller

The enhanced interrupt controller operation may be turned on for MPC565/563/561. In this mode the 32 IMB interrupt levels will be latched by USIU using eight IMB interrupt lines and two lines of ILBS via time multiplexing scheme defined by the UIMB module. In addition to the IMB interrupt sources the external interrupts and timer interrupts are available by the same way as in the regular scheme.

Table 12.2 is a summary of interrupt sources and levels for MPC565/563/561 and according *ISR/PRIORITY* attribute value:

Table 12.2   ISR Priority and Interrupt Sources for MPC565/563/561

ISR PRIORITY	Interrupt sources
47	EXT_IRQ 0
46	Level 0
45	IMB_IRQ 0
44	IMB_IRQ 1
43	IMB_IRQ 2
42	IMB_IRQ 3
41	EXT_IRQ 1
40	Level 1
39	IMB_IRQ 4
38	IMB_IRQ 5
37	IMB_IRQ 6
36	IMB_IRQ 7
35	EXT_IRQ 2
34	Level 2
33	IMB_IRQ 8
32	IMB_IRQ 9
31	IMB_IRQ 10
30	IMB_IRQ 11
29	EXT_IRQ 3
28	Level 3
27	IMB_IRQ 12

Table 12.2   ISR Priority and Interrupt Sources for MPC565/563/561

ISR PRIORITY	Interrupt sources
26	IMB_IRQ 13
25	IMB_IRQ 14
24	IMB_IRQ 15
23	EXT_IRQ 4
22	Level 4
21	IMB_IRQ 16
20	IMB_IRQ 17
19	IMB_IRQ 18
18	IMB_IRQ 19
17	EXT_IRQ 5
16	Level 5
15	IMB_IRQ 20
14	IMB_IRQ 21
13	IMB_IRQ 22
12	IMB_IRQ 23
11	EXT_IRQ 6
10	Level 6
9	IMB_IRQ 24
8	IMB_IRQ 25
7	IMB_IRQ 26
6	IMB_IRQ 27
5	EXT_IRQ 7
4	Level 7
3	IMB_IRQ 28
2	IMB_IRQ 29
1	IMB_IRQ 30
0	IMB_IRQ 31

For more detailed information about MPC565/563/561 enhanced interrupt controller see MPC565/563/561 User's Manual.

ISRs which have interrupt source beside interrupt controller in case Enhanced Interrupt Controller using have to have *PRIORITY* 48 for MPC565/563/561.

- *RESOURCE* (reference)
  The reference specifies resource accessed by the ISR. There can be several *RESOURCE* references. This parameter can be omitted. The reference can not be defined if *CATEGORY* value is equal to 1.

- *ACCESSOR* (ENUM)
  The attribute is used to define type of usage for the message. The ISR uses the accessor for definition of references to sent or received messages.

- *MESSAGE* (reference)
  The reference specifies the message to be sent or received by the task. This parameter is a single reference, it has to be defined.

- *ACCESSNAME* (STRING)
  The attribute defines the reference which can be used by the application to access the message data. This attribute has to be defined. The local copy of the message shall be defined with this name in application code.

Resource Definition

This object is intended for the resource management. The resource *Ceiling priority* is calculated automatically on the basis of information about priorities of tasks using the resource. The keyword RESOURCE is used for this object type. Section "Resource Management" describes resource concept in OSEK. The syntax of the resource object is as follows:

```
RESOURCE <name of resource> {
  RESOURCEPROPERTY = <STANDARD / LINKED /  INTERNAL> {
    LINKEDRESOURCE = <name of RESOURCE>
  };
};
```

- *RESOURCEPROPERTY* (ENUM)

  The attribute specifies a property of the resource. The *STANDARD* value corresponds to a normal resource which is not linked to another resource and is not an internal resource. The *LINKED* value corresponds to a resource linked to another resource with the property *STANDARD* or *LINKED*. The *INTERNAL* value is appropriate to an internal resource which cannot be accessed by the application.

Performance decreases if the RESOURCE object with the INTERNAL value of the RESOURCEPROPERTY subattribute is defined.

- *LINKEDRESOURCE* (reference)

    The attribute specifies the resource to which the linking shall be performed. The OS System Generator resolves chains of linked resources. This reference should be defined only if the value of the *RESOURCEPROPERTY* attribute is *LINKED*.

Event Definition

This object is intended for the event management. The event object has no references. The keyword EVENT is used for this object type. Section "Events" describes events in OSEK. The syntax of the event object is as follows:

```
EVENT <name of EVENT> {
  MASK = <integer / AUTO>;
};
```

Attribute

The object has one standard attribute:

- *MASK* (UINT64)
    The event is represented by its mask. The event mask is the number which range is from 1 to 0xFFFFFFFF, preferably with only one bit set. The other way to assign event mask is to declare it as *AUTO*. In this case event masks will be assigned automatically according to their distribution among the tasks.

Counter Definition

This object presents OSEK Operating system counters. Attributes of this object type define counter properties. A counter has no references, it is referenced to by other object. The keyword COUNTER is used for this object type. OSEK counters are described in section "Counters and Alarms". The syntax of the counter object is:

```
COUNTER <name of COUNTER> {
```

```
                    MINCYCLE = <integer>;
                    MAXALLOWEDVALUE = <integer>;
                    TICKSPERBASE = <integer>;
               };
```

Attributes

The object has the following standard attributes:

- ***MINCYCLE*** (UINT32)
  The attribute specifies the minimum allowed number of counter ticks for a cyclic alarm linked to the counter. (In fact, this parameter has a sense only for systems with extended OS status since it is checked in this case only.)

- ***MAXALLOWEDVALUE*** (UINT32)
  The attribute defines the maximum allowed counter value. After the counter reaches this value it rolls over and starts count again from zero.

- ***TICKSPERBASE*** (UINT32)
  The number of ticks that are required to reach a counter-specific value. This value cannot be derived automatically from other counter related attributes. The interpretation is up to the user.

Alarm Definition

This object presents alarms. Links with other system objects are defined via references. The referenced counter and task must be already defined. The keyword ALARM is used for this object type. See section "Alarms" for information about alarms.

The syntax of an alarm object is as follows:

```
ALARM <name of ALARM> {
  COUNTER = <name of COUNTER>;
  ACTION = <SETEVENT / ACTIVATETASK / ALARMCALLBACK> {
    TASK = <name of TASK>;
    EVENT = <name of EVENT>;
    ALARMCALLBACKNAME = <string>;
  };
  AUTOSTART = <TRUE / FALSE> {
    ALARMTIME = <integer>;
    CYCLETIME = <integer>;
```

```
    APPMODE = <name of APPMODE>;
  };
};
```

Attributes

The object has the following attributes:

- *COUNTER* (reference)
  The reference specifies the assigned counter. An alarm shall be assigned to a particular counter to have an ability to operate. Only one counter has to be assigned to the alarm.

- *ACTION* (ENUM)
  The attribute defines which type of task notification is used when the alarm expires. For one alarm only one action is allowed. If the *ACTION* attribute is defined as *ACTIVATETASK*, the *TASK* reference defines the task to be activated when the alarm expires. If the *ACTION* attribute is defined as *SETEVENT*, then the *TASK* references defines the task to be activated, and *EVENT* references defines the event to be set when the alarm expires. If the *ACTION* attribute is defined as *ALARMCALLBACK*, then the *ALARMCALLBACKNAME* subattribute specifies the name of the callback routine called when the alarm expires.

- *ALARMCALLBACKNAME* (STRING)

  The attribute specifies the name of the callback routine called when the alarm expires. The parameter should be specified if the *ACTION* attribute is set as *ALARMCALLBACK*.

- *TASK* (reference)
  The reference to a task which is to be notified via activation or event setting when the alarm expires.

- *EVENT* (reference)
  The reference specifies the event mask to be set when the alarm expires. The event is considered as an inseparable pair of the task and the event belonging to this task, so the reference to the task which owns the events shall be also defined for this alarm.

  The reference shall be defined if the *ACTION* value is *SETEVENT*.

- *AUTOSTART* (BOOLEAN)

  The attribute defines whether an alarm is started automatically at system start-up depending on the

application mode. If the alarm should be started at the system start-up, the value is set to TRUE otherwise the value is set to FALSE. When the *AUTOSTART* attribute set to *TRUE*, the *ALARMTIME, CYCLETIME*, and *APPMODE* parameters should be defined.

- *ALARMTIME* (UINT32)

  The attribute defines the time when the alarm shall expire first. The attribute should be be defined if the *AUTOSTART* attribute is set to *TRUE*.

- *CYCLETIME* (UINT32)

  The attribute defines the cycle time of a cyclic alarm. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*.

- *APPMODE* (reference)

  The attribute defines an application mode for which the alarm shall be started automatically at system start-up. The attribute should be defined if the *AUTOSTART* attribute is set to *TRUE*. It may be ommited if only one *APPMODE* is defined.

Message Definition

This object is intended to present either a Unqueued or an Queued message. Attributes of this object type define message properties. Links with other system objects are defined via references. The keyword MESSAGE is used for this object type. Messages concept is described in section "Communication". The syntax of a message object definition is as follows:

```
MESSAGE <name of MESSAGE> {
  TYPE = <QUEUED / UNQUEUED>;
    QUEUEDEPTH = <integer>;
  CDATATYPE = <string>;
  ACTION = <ACTIVATETASK / SETEVENT / CALLBACK / FLAG / NONE> {
    TASK = <name of TASK>;
    EVENT = <name of EVENT>;
    CALLBACKNAME = <string>;
    FLAGNAME = <string>;

  };
```

```
};
```

Attributes

The object has the following standard attributes:

- *TYPE* (ENUM)
  The attribute specifies type of the message. In accordance with the OSEK specification there are two types of messages: unqueued and queued. Queued message data is buffered and consumed by receive operations. Unqueued message data is not consumed and overwritten each time.

- *QUEUEDEPTH* (UINT64)
  The attribute specified if the message has a queue. If used for internal communication, the COM conformance class will be CCCB. The parameter can be defined only if the *TYPE* attribute is QUEUED.

- *CDATATYPE* (STRING)
  The attribute defines the data type of the message item. The message item can have the own type which has to be any C data type. Any ANSI-C type specifier is allowed. It is the standard C type identifier - *char, int, float, double* with any type modifiers (*signed, unsigned, short, long*) and also structure or union specifier (starting *struct or union*), enum specifier (starting *enum*), *typedef* name (any valid C-language identifier) enclosed in the double quotas. To use an array of standard C-language type the user must define the new type via *typedef* operator. In case of user's defined data types or enumerations such definitions must be in the user's code before using files produced by SG.

- *ACTION* (ENUM)
  The attribute defines which type of task notification is used when the message arrives. Only one action per message is provided.

- **TASK** (reference)
  The reference specifies the task which shall be notified when the message arrives.

  This reference shall be defined only if the value of the *ACTION* attribute is *ACTIVATETASK* or *SETEVENT*.

- *EVENT* (reference)
  The reference specifies the event which is to be set when the message arrives. The event is considered as an inseparable pair of the task and the event belonging to this task, so the reference

to the task which owns the events shall be also defined for this message.

This reference shall be defined only if the value of the *ACTION* attribute is *SETEVENT.*

- **CALLBACKNAME** (STRING)
  The attribute defines the name of function to call as an action when the message arrives. It shall be defined only if the value of the *ACTION* attribute is *CALLBACK.*

- **FLAGNAME** (STRING)
  The attribute defines the name of the flag that is set when the message is sent. It shall be defined only if the value of the *ACTION* attribute is *FLAG.*

Application Modes Definition

It is possible to introduce different application modes inside one CPU container by means of objects named *APPMODE.* Each *APPMODE* object defines a set of autostarted Tasks and Alarms for an OSEK OS application mode.

No standard attributes are defined for the APPMODE object. At least one APPMODE object has to be defined in a CPU.

The syntax of an application mode object definition is as follows:

```
APPMODE <name of mode>;
```

OSEKturbo OS supports up to **8** application modes.

COM Definition

The COM object represents OSEK communication subsystem properties on CPU. Only one COM object must be defined on the local CPU.

The syntax scheme of COM object is as follows:

```
COM <name of COM> {
    USEMESSAGERESOURCE = <TRUE / FALSE>;
    USEMESSAGESTATUS = <TRUE / FALSE>;
```

```
};
```

Attributes

The object has the following standard attributes:

- *USEMESSAGERESOURCE* (BOOLEAN)
  The attribute specifies if the message resource mechanism is used.

- *USEMESSAGESTATUS* (BOOLEAN)
  The attribute specifies if the message status is available.

NM Definition

The NM object represents the local parameters of the network management subsystem on CPU. No attributes are defined for the NM objects.

OSEKturbo Performance Dependency

The following attributes of OS object directly affect OS performance:

- *STATUS* – if this attribute is set to EXTENDED, then the system performance is decreased because OS spends additional time for checking. See "System Services" for more detailed services description.

- *StackOverflowCheck* – if this attribute is TRUE, then the stack overflow conditions are checked when each rescheduling occurs and in each ISR, thus significantly decreasing OS performance.

- *DEBUG_LEVEL* – if this attribute is not equal 0, then *StackOverflowCheck* is (implicitly) turned on. Additional resource management is affected: resources state manipulations in code added.
  If this attribute is greater than 1 additional code to support tracing is added into all OS services.

- *InterruptDispatcher* – if this is not set to None, then additional code is executed before entering any ISR. The user should set *InterruptDispatcher* to OneLevel, MultiLevel or Enhanced (for MPC565/563/561) only when it is necessary.

- *UnorderedExceptions* – if this is not set to TRUE, then additional code is executed before entering any ISR. The user should set *UnorderedExceptions* to TRUE only when it is necessary for debugging.

- *FloatingPoint* – if this attribute is set to TRUE, then OS performance is decreased because all Floating Point (FP) registers are saved/restored at task switch.

- *FastTerminate* – if this attribute is FALSE, then the OS has to save the state at the starting point of each Task and restore it in *TerminateTask* (or *ChainTask*). If *FastTerminate* is set to TRUE, then the OS performance is increased, but all calls to *TerminateTask* or *ChainTask* services must be done from Task body function level. This attribute is applicable only for BCC1.

- *FastScheduler* – if this attribute is set to TRUE, then the OS performance is increased because OS places some of it's variables into the CPU registers. See <u>"Low-Power Mode,"</u> for details of registers usage.

- *PRETASKHOOK* and *POSTTASKHOOK* – if they are set to TRUE then the system performance is decreased even if corresponding routines (*PreTaskHook* and *PostTaskHook*) are empty because the OS calls them at each task switch.

- *ERRORHOOK* – if set to TRUE then the system performance is decreased if *STATUS* or *StackOverflowCheck* are also set to TRUE.

- *USEGETSERVICEID* and *USEPARAMETERACCESS* - if any of them is set to TRUE and *ERRORHOOK* is set to TRUE too then system performance is decreased.

The following attributes of RESOURCE object directly affect OS performance:

- *RESOURCEPROPERTY* – if this attribute is set to INTERNAL, then the system performance is decreased because OS spends additional time for setting and releasing priorities of tasks when switching occurs.

The best system performance may be achieved by leaving all mentioned in this section attributes with their default values, except *FastTerminate* and *FastScheduler,* which shall be explicitly set to TRUE.

13

Building of Application

This chapter contains information on how to build an user's application using OSEK OS. It also describes memory requirements.

This chapter consists of the following sections:

Application Structure

An application developed on the OSEK Operating System basis has a defined structure. An application consists of the Operating System kernel and several user's tasks and ISRs, which interact with the kernel by means of system services and internal mechanisms. ISRs receive control from hardware interrupt sources via the vector table. Tasks are controlled by the scheduler. They may use all means for intertask communications granted by OSEK OS to pass data and synchronize each other.

Tasks and ISRs are considered as system objects. Resources, messages, counters, and alarms are also considered as system objects, because they are controlled by the Operating System. An application typically also has configuration tables for different system objects, task stacks and other entities. To create an application, the user should develop the desired application structure with all necessary objects and define interactions between them.

All global Operating System properties, system objects and their parameters are defined by the user statically and cannot be redefined at run time. Special application configuration file is designed to perform such definition and the special tool that processes this file. See "System Configuration". After processing, files with system object descriptors are created automatically. These

files provide the code for all required ROM and RAM structures, arrays, tables, variables, etc. for all system objects defined in the configuration file. Memory allocation is performed during start-up procedure.

Action Sequence to Build an Application

To build an application using the OSEK Operating System the user should perform a set of actions. These actions are relatively simple since the most important requirement is a clear understanding of the application's algorithm. The actions include creating the application configuration file, processing this file by the System Generator, writing the user's source code, compiling all files and linking the application files together with the OSEK OS code. This process is shown in Figure 13.1.

Figure 13.1   Application Building Process
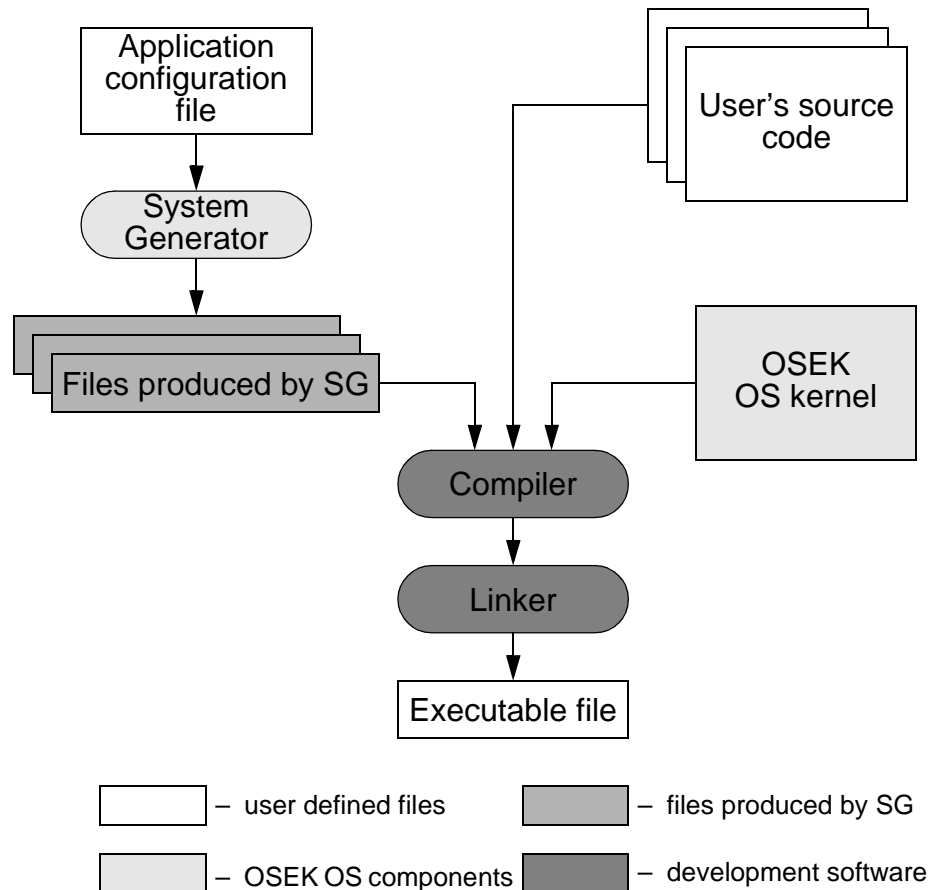
Application Configuration

Applications built using OSEK OS are configured statically via the special configuration file written in OIL. "System Configuration" describes the structure of such file and "System Objects Definition" describes all possible statements in detail. This configuration file defines system specific parameters as well as system objects. Such a file can have any extension and the extension ".oil" is suggested by default.

The configuration file has to be processed by the special utility named System Generator (SG). This utility is delivered as one of the parts of the OSEK Operating System. This tool runs as a 32-bit

console application for Windows 2000/98 and produces header and source files.

The following command is used to run SG:

```
sysgen [-options] oil_file
```

The following command line options are intended to control SG:

Table 13.1   System Generator Command Line Options

Option	Description	Default value
-b	Defines a format of messages output as tree-like form	No
-c<name>	Defines <name> as the output c-data file name	Input OIL file name with ".c" extension
-f<name>	Defines <name> of command file for SysGen command line parameters	No
-h<name>	Defines <name> as the output header file name	Input OIL file name with ".h" extension
-i<path>	Defines <path> as the path for include files	No
-n<CPU>	Defines <CPU> as the name of CPU for which output files are generated	First CPU in the file
-p<name>	Defines <name> as the OS property file name	"osprop.h" in the source OIL file directory
-O<version>	Specifies ORTI version	2.1
-o<name>	Defines <name> as the ORTI information file name	Input OIL file name with ".ort" extension
-t	Only verification for OIL input file. No output files shall be generated	No
-v	When this option is defined, SysGen shall output versions of all its components	No
-w -w<message identifier>	Suppresses all warning messages Suppresses warning message defined by <message identifier	No
-A	Forces to generate absolute paths in include directives of configuration files instead of relative one	No

In addition to command line option the OSB_INCLUDE_DIR environment variable can be used to specify the set of directories to search for include files.

The SG utility produces three types of standard C-language files which are to be compiled and linked together with OS kernel code and user's source code:

1. The header file which describes the current properties of the operating system. This file contains the preprocessor directives `#define` and `#undef`. This file is used at compile time to build the OS kernel with the specified properties. The default filename is "osprop.h" but the user can assign another name (see <u>"Source Files"</u>).

2. The application configuration header file which contains definitions of all system objects, data types, constants and external declarations of variables which are needed to describe system objects. This file is used to compile application files. By default, System Generator uses the input file name for this output file with ".h" extension.

3. The source file which contains initialized data and memory allocation for system objects. This file is compiled with "osprop.h" and other header files and then linked together with other application and OS files. By default, System Generator uses the input file name for this output file with "c" extension.

NOTE As a rule, the user is not allowed to edit files produced by the System Generator. It may lead to data inconsistency, compilation errors or unpredictable application behavior.

Source Files

OSEK Operating System is delivered to the user as a set of source files. Header and source files of the Operating System are located in the predefined directories after OSEK OS installation. Paths to these directories have to be provided by the user.

The OS source code is compiled and linked together with other application's files. The header file "osprop.h" describing system properties defines which functionality will have the OS kernel in run time. Generally, changes in OIL file result in "osprop.h" modification and require recompilation of OS files. However some of object attributes do not affect "osprop.h" contents, see <u>"OS Object Files Dependency"</u> for details.

For More Information: www.freescale.com

This file must be included in all user's and OS' source files. Since the user can specify another name for this file the special macro *OSPROPH* is designed to substitute the name. After OS property file the file osapi.h shall be included to make OS services visible to the user application. Then OS configuration file (cfg.h), which contains declarations of all OS objects shall be included. Macro *OSCFGH* may be used to substitute a name of configuration header file. The following code could be used in all user's files:

```
#if !defined (OSPROPH)
#include  "osprop.h"
#else     /* !defined (OSPROPH) */
#include  OSPROPH
#endif    /* !defined (OSPROPH) */

#include  "osapi.h"

#if  !defined(OSCFGH)
#include  "cfg.h"
#else     /*!defined(OSCFGH)*/
#include  OSCFGH
#endif    /*!defined(OSCFGH)*/
```

The compiler command line (see "Compiling and Linking") in this case should have the options like this:

```
-dOSPROPH="<property_name>".
-dOSCFGH="<config_name>".
```

Where <property_name> is the name of the file with system properties definitions and *<config_name>* is the name of the file with system configuration definitions

But the user is allowed to use some other method to include the property definition and configuration header files in his/her source code.

Among other files SG generates configuration C-file containing definitions and initialization of OSEK OS configuration data declared in corresponding configuration header file. Configuration C-file is a separate module, however, in some particular OSEK OS configurations, it could contain references to

user defined data and structures (e.g. user's message structure types). This requires a method to provide SG generated configuration C-file with such user defined types and data declarations. Thus SG generates the following code in configuration C-file:

```
#if defined(APPTYPESH)
#include APPTYPESH /* user's header file */
#endif /* defined(APPTYPESH) */
```

The compiler command line (see "Compiling and Linking") in this case should have the option like this:

```
-dAPPTYPESH="<filename>".
```

`<filename>` is the name of the file with user defined structures and data declarations.

In the example below one data type and variable are defined by the user which are referenced in files generated by SG. Variable are defined in the user's file "user.c" and referenced in the produced file "cfg.c". The data type is defined in the user's file "user.h" and referenced in the produced file "cfg.c". The user's code can be the following:

USER.H file:

```
typedef struct tagMSG MSGTYPE;
struct tagMSG
{
  TickType timeStamp;
  int x;
};
extern MSGTYPE MsgA;
```

USER.C file:

```
#include "user.h"   /* include user defined data type */
...
MSGTYPE MsgA;        /* user defined variables */
...
```

CFG.C file (generated by SG):

```
...
#if defined(APPTYPESH)
#include APPTYPESH /* user's header file */
#endif /* defined(APPTYPESH) */
...
/* SG generated code referring to user's type and data */
...
```

The compiler command line has the following option:

```
-dAPPTYPESH="USER.H".
```

Other variants are also possible.

The code of user's tasks and functions should be developed according to common rules of the C language. But some exceptions exist:

- The keyword *TASK* and *ISR* should be used to define a task and ISR correspondingly;
- For objects controlled by the OSEK Operating System the data types defined by the system must be used. The data types are described at the end of previous sections and in "System Services".

Compiling and Linking

When all needed header and source files are created or produced by the System Generator an application can be compiled and linked (for details see "PowerPC Platform-Specific Features").

Linking process is controlled by the typical linker directive file.

OS Object Files Dependency

The OS object files are recompiled when content of the OS property file is changed. The OS configuration depends on many parameters defined in OIL file but there exists a set of parameters which can be changed without necessity to recompile OS files (or rebuild OS library). Note that number of objects affects constants defined in OS

property file, so adding or deleting an object will cause OS recompiling.

The configuration attributes which do not require recompiling of OS are listed below.

OS attributes:

- *MessageCopyAllocation*
- *TimeScale* subattributes:
    - *ScalePeriod*
    - *TimeUnit*
    - *Step* subattributes: *StepNumber*, *StepTime*, *TASK*

TASK attributes:

- *PRIORITY*
- *AUTOSTART*
- *RESOURCE*
- Number of *ACCESSOR* definition and all subattributes of *ACCESSOR*: *MESSAGE*, *WITHOUTCOPY*, *ACCESSNAME*
- *STACKSIZE*

ISR attributes:

- Number of *ACCESSOR* definition and all subattributes of *ACCESSOR*: *MESSAGE*, *ACCESSNAME*

COUNTER attributes:

- *MINCYCLE*
- *MAXALLOWEDVALUE*
- *TICKSPERBASE*

ALARM attributes:

- *COUNTER*
- The *ACTION* subattributes *TASK*, *EVENT,*
  ALARMCALLBACKNAME

MESSAGE attributes:

- *CDATATYPE*

- The following *ACTION* subattributes:
  - *TASK*
  - *EVENT*
  - *CALLBACKNAME*

Sample Application

In <u>"Sample Application"</u> the code of an OSEK OS based application is provided. This code is a simple demonstration of Operating System mechanisms. It also demonstrates how to write the configuration file and source code.

14

PowerPC Platform-Specific Features

This chapter discusses special OSEK OS features for different MCU types and issues connected with porting applications to these MCUs.

This chapter consists of the following sections:

- Compiler-Specific Features
- General and Special Purpose Registers Usage
- Stack Size
- MPC555 Features
- MPC565/563/561 Features

Compiler-Specific Features

The following tools should be used to build OSEK OS applications:

- D-CC v.5.0a for PowerPC
- Metrowerks C/C++ Compiler for Embedded PowerPC version 3.0.4.

Used Library Functions

OSEKturbo OS itself does not use any functions from compilers run-time libraries except *memcpy* and *memset.*

Compiler Issues

Installation procedure defines environment variable values in batch files which is used for sample compilation. If they were not set during installation the user should do it manually to compile sample. These variables are the following:

For More Information: www.freescale.com

OSEKDIR = [path] – path to the OSEK directory

SYSGENDIR = [path] – path to the System Generator directory

CWDIR = [path] – path to the CodeWarrior compiler

DIABDIR = [path] – path to Diab Data compiler

See the makefile in the `SAMPLE\STANDARD` directory for additional information.

User should use special compiler options for compiling OSEK OS modules.

NOTE	It is recommended to use compiler/linker options list as it is set in the sample makefiles. Using of the other compiler options may lead to possible compiling/linking problems. Therefore, in order to escape such problems and be sure that proper code to be generated please refer to makefiles for exact option list.

Options for Diab Data Software

Options for D-CC Compiler

The following D-CC compiler options should be used for compiling all OSEK OS C-source modules:

```
-c -D OSDIABPPC -tPPC555E<FPsupport>:simple -Xnested-interrupts
-Xansi -O -XO -DOSINTMEMBASE=<intmembase>
```

Where <FPsupport> is H (hardware), S (software), N (none); <intmembase> is the base address of the internal memory. If <FPsupport> is set to H (hardware) then OS configuration option *FloatingPoint* shall be set to TRUE.

NOTE	In some complicated cases D-CC generates a wrong code due to the optimization error. If this is a case the option "-Xkill-opt=0x00010000" (global common subexpression elimination) should be added into the command line.

Please see D-CC compiler User's Manual for details on compiler options.

Options for D-CC Assembler

The following compiler options should be used for D-CC assembler:

```
-tPPC555E<FPsupport>
```

Where <FPsupport> is H (hardware), S (software), N (none). If it is set to H (hardware) then the OS configuration option *FloatingPoint* shall be set to TRUE.

Options for Metrowerks Software

Options for Metrowerks C/C++ Compiler for Embedded PowerPC

The following Metrowerks C/C++ compiler options should be used for compiling all OSEK OS C-source modules:

```
-c -fp <FPsupport> -proc 555 -ansi off -opt all -Cpp_exceptions
off -DOSINTMEMBASE=<intmembase>
```

Where <FPsupport> is hard (hardware), soft (software) or none (none); <intmembase> is the base address of the internal memory. If <FPsupport> is set to hard (hardware) then the OS configuration option *FloatingPoint* shall be set to TRUE.

Please see Metrowerks compiler User's Manual for details on compiler options.

Options for Metrowerks Assembler for Embedded PowerPC

The following compiler options should be used for Metrowerks assembler:

```
-proc 555
```

Options for Metrowerks Linker for Embedded PowerPC

```
-fp <FPsupport>
```

Where <FPsupport> is hard (hardware), soft (software) or none (none). If it is set to hard (hardware) then the OS configuration option *FloatingPoint* shall be set to TRUE.

Link OS Code into Special Segment

In both compilers there is possibility to redirect the OS code into special memory segment using linker features. This may be useful for more precise memory placement. The following fragments of the Linker Command Files contains strings for OS memory section definition. Section names and their addresses may be changed according to the user's discretion.

Linker Command File for Diab-CC

```
MEMORY
{
  ram:org = 0x00003000, len = 0xB000
           /*address and size for OS segment */
  os_rom:  org = 0x0000E000, len = 0xB000
  vect:org = 0x00000000, len = 0x2000
}
SECTIONS
{
  GROUP : {
      /* First take all code and constants from all OS objects */
    .ostext (TEXT)   : {$(oilname).obj(.text),
               os.obj(.text), ostsk.obj(.text), osisr.obj(.text),
                            osalm.obj(.text), osctr.obj(.text),
                            osres.obj(.text), osevt.obj(.text),
                            osmsg.obj(.text), ossch.obj(.text),
                            osset.obj(.text), ostrg.obj(.text)}
    /* Next take all small CONST data */
    .ossdata2 : {$(oilname).obj(.sdata2),
        os.obj(.sdata2), ostsk.obj(.sdata2), osisr.obj(.sdata2),
                 osalm.obj(.sdata2), osctr.obj(.sdata2),
                 osres.obj(.sdata2), osevt.obj(.sdata2),
                 osmsg.obj(.sdata2), ossch.obj(.sdata2),
                 osset.obj(.sdata2), ostrg.obj(.sdata2)}
  } > os_rom
...
}
```

Linker Command File for Metrowerks Linker for Embedded PowerPC

```
MEMORY
{
  ram:org = 0x00003000, len = 0x0000B000
           /*address and size for OS segment */
  os_rom:  org = 0x0000E000, len = 0x0000B000
  vect:org = 0x00000000, len = 0x00002000
}

SECTIONS
{
   /* The first group contains code and constant data exclude OS
code and constants */
  GROUP : {
    .text (TEXT)   : {
      *(.init) *(.fini) *(.text) *(.rodata) *(.rdata)
    }
    .sdata2 (DATA) : {}
    .rodata (CONST) : {}
    extab (CONST) : {}
    .dtors (CONST) : {}
    .ctors (CONST) : {}
    extabindex (CONST) : {}
  } > ram
  /* The second group contains OS code and constant data */
  GROUP : {
    .ostext (TEXT)   : {os.obj ostsk.obj osisr.obj osalm.obj
osctr.obj   osres.obj osevt.obj osmsg.obj ossch.obj osset.obj
ostrg.obj}
    /* Next take all CONST data */
    .osrodata (CONST) : {$(oilname).obj os.obj ostsk.obj
osisr.obj osalm.obj osctr.obj osres.obj osevt.obj osmsg.obj
ossch.obj osset.obj ostrg.obj}
  } > os_rom
...
}
```

General and Special Purpose Registers Usage

The Special purpose register 2 (SPRG2) is used to save/restore the basic stack pointer.

If *FastScheduler* attribute is set to TRUE then the general purpose registers r14, r15, r16 and r17 are used by OSEKturbo OS for global variables. The register r17 is used only when there are more than 32 tasks configured.

NOTE   In case when *FastScheduler* is set to TRUE and some source files or libraries do not include osapi.h header file the compiler shall be prevented from the usage of these registers. It may be done by including the file hwspec\osregs.h into source files. If modification of source files is not possible then the compiler command line option shall be used. Table 14.1 displays command line options for CodeWarrior and Diab Data compilers.

Table 14.1   Register Reservation in User's Applications

Compiler	Command Line
Metrowerks	-AddIncl="osregs.h"
Diab-CC	-i=osregs.h

Stack Size

Generally, the recommended minimal task stack size equals:

- 200 bytes for simplest preemptive tasks
- 300 bytes for more complex preemptive tasks (with task, event, message manipulation)
- 400 bytes for complex preemptive tasks (task, event, message manipulation, interrupts)

Stack Usage for OS Services

The task stack usage depends on services called in the task and usage of ISRs in the application. Data from the tables below can be used for the rough estimation of task and ISR stack size.

The values given in the Table 14.2 are measured for the typical application configuration for OSEKturbo OS/MPC5xx compiled with CodeWarrior and Diab Data compilers. OSEK is a highly scalable RTOS and most of these values depend on configuration

Table 14.2   Stack Usage for OS Services

System service	CC	Stack in bytes	
		CodeWarrior	Diab-CC
ActivateTask with rescheduling	ECC1	48	64
TerminateTask	BCC1	32	32
TerminateTask	ECC1	32	32
ChainTask	BCC1	32	32
ChainTask	ECC1	32	32
Schedule with rescheduling	ECC1	32	32
GetTaskId	BCC1	0	0
GetTaskState	BCC1	0	32
EnableAllInterrupts	BCC1	0	0
DisableAllInterrupts	BCC1	0	0
ResumeAllInterrupts	BCC1	0	16
SuspendAllInterrupts	BCC1	0	16
ResumeOSInterrupts	BCC1	0	16
SuspendOSInterrupts	BCC1	0	16
GetResource	BCC1	16	16
ReleaseResource	BCC1	16	32
SetEvent with rescheduling	ECC1	32	48
ClearEvent	ECC1	0	16
GetEvent	ECC1	0	0
WaitEvent wuth rescheduling	ECC1	32	32
SendMessage	BCC1	32	48
ReceiveMessage	BCC1	32	32
InitCounter	BCC1	0	16
CounterTrigger with rescheduling	BCC1	32	32
GetCounterValue	BCC1	0	16
GetCounterInfo	BCC1	0	0
GetAlarmBase	BCC1	0	0
SetRelAlarm	BCC1	32	48
SetAbsAlarm	BCC1	32	48
CancelAlarm	BCC1	16	32
GetAlarm	BCC1	0	32

Table 14.2   Stack Usage for OS Services

System service	CC	Stack in bytes	
		CodeWarrior	**Diab-CC**
StartTimeScale with resceduling	BCC1	80	96
StopTimeScale	BCC1	0	0

The Table 14.3 shows stack usage for empty Tasks and ISRs and for System Timer ISR.

Table 14.3   Stack Usage for Empty Tasks/ISRs

Operation	CC	CodeWarrior	Diab-CC
Empty BASIC task[a]	BCC1	80	96
EXTENDED task with WaitEvent inside endless loop	ECC1	54	54
Empty ISR category 1	BCC1, ECC1	48	80
ISR category 2, Task stack	ECC1	96	100
Empty ISR category 2, ISR stack,	ECC1	22	26
SystemTimer ISR, Alarm activated (ISR Stack)	ECC1	24	28

[a]   Including ActivateTask stack usage

To estimate the approximate size of a task stack, list the function calls, take the one with the largest stack usage and add amount used by empty task to the stack usage. If there are ISRs, add stack size needed for ISR.

MPC555 Features

Programming Model

MPC/5200 can operate at either of two privilege levels. Supervisor level is more privileged than user level.

OSEK OS uses supervisor programming model, so start-up code should provide correct initialization. OSEK OS always works in the big-endian byte ordering mode.

For More Information: www.freescale.com

Target Hardware Initialization

OSEK OS does not provide target hardware (except used by system timers) initialization possibilities, therefore target hardware initialization (memory used, PowerPC cache settings, etc.) should be performed before system startup.

Interrupt Vector Table

If the *InterruptDispatcher* attribute has value "OneLevel", "MultiLevel"or "Enhanced" OS/MPC provides the external interrupt (sharing 0xYYY00500 interrupt vector) distinction and the appropriate interrupt handler call. In this case OS/MPC snaps up the external interrupt exception, makes required save/restore operations and calls appropriated interrupt handler. If the *InterruptDispatcher* attribute has "None" value and user needs to add ISRs he has to modify the vector table. Also vector table should be modified if user wants to use any other vectors besides external interrupt and decrementer vectors.The interrupt vector table is defined in the file "vector.c" which is delivered with the OSEK Operating System and located in the HWSPEC directory. This file is the example of the interrupt vector table coding.

The user should copy this file into the project directory and then modify it as needed.

For ISRs of category 2 the following interrupt table entries may be used:

Let's assume, that ISR is defined via the following statement in configuration OIL-file:

```
ISR <nameISR> {
    ...      // ISR category should be defined here
};
```

In this case, interrupt vector table should contain absolute branch (MPC "ba" instruction) to <nameISR> interrupt handler.

```
asm( "ba <nameISR>" );
```

And ISR handler definition should look like the following in the application source file:

```
ISR( <nameISR> ) {
    ...        // ISR body should be defined here
};
```

The user shouldn't remove references to System and Second Timer ISR's if they are used by application.

Please see `vector.c` located in `HWSPEC` directory for details.

Low-Power Mode

The user can configure OSEK OS to use low-power mode when system runs idle – *HCLowPower* configuration option (see "Global System Attributes"). If the *HCLowPower* is set to TRUE the system enters Doze-high mode in idle loop (when there is no running task).

The user is able to implement own low-power mode logic in the framework of *IdleLoopHook*. If this is a case it is recommended to set *HCLowPower* to FALSE. (see "Hook Routines Related Attributes").

MSR Bits Manipulation

The user should not manipulate with MSR general-purpose bits like FP, ME, FE0, FE1, IP during run-time. If it is necessary to set MSR ME bit (for example), then it should be set before OS start-up.

Implementation Background

OSEK OS task context contains all information needed to provide both ordinary task scheduling and scheduling after ISR.

Using Floating Point

OSEKturbo OS does not save/restore floating point registers during system startup/shutdown.

By default, OS does not save/restore floating point registers during context switch. Therefore, if it is necessary to perform floating-point calculations inside OSEKturbo OS application, *FloatingPoint* option should be used.

WARNING!   It is not allowed to perform floating-point calculations inside ISR handlers.

The following statements should be used to configure floating-point support in OSEK OS:

```
OS <name> {
...
    FloatingPoint = <TRUE/FALSE>;
...
};
```

The *FloatingPoint* configuration option turns floating-point support. This option has the type *BOOLEAN*. The possible values are the following:

- TRUE - floating-point registers are saved/restored during context switch, FP bit in MSR is always set during task's execution.

- FALSE - floating-point registers are not saved/restored during context switch, the state of FP bit is indeterminate.

NOTE   If the FloatingPoint option has the TRUE value then the <FPsupport> compiler option has to be not N (none).

Nested Interrupts

According to the OSEK OS v.2.2 specification there are no any services which enables interrupts directly. Therefore nested interrupts with the same hardware levels can not occur. Application must not enable interrupts in ISR using direct manipulation of CPU registers – it may cause an unpredictable application behavior (there is an exception for this rule - see <u>Enhanced Interrupt Controller</u>.) Interrupts of different levels are processed in usual way and may be nested.

For multilevel emulation and enhanced modes nested interrupts with different priorities are allowed.

For More Information: www.freescale.com

Interrupt Dispatcher

OSEKturbo OS∕MPC5xx provides threelevels of the interrupt dispatching:

- None, when interrupt dispatching is not supported
- OneLevel, when one interrupt level for external interrupts is supported
- MultiLevel, when multilevel interrupt emulation mode is supported

NOTE   There are two interrupt levels even in case of None and OneLevel Interrupt dispatcher: maskable and unmaskable interrupts. The words "interrupts level" means maskable external interrupts.

External Interrupts Processing Background

All External Interrupts share one vector offset 0xYYY00500. Interrupt "sources" originate interrupt service requests. These sources include interrupts from internal sources (such as the PIT and RTC), from IMB3 module and from external pins IRQ[0:7]. Each source is identified by a "level". Some interrupt sources have fixed levels. Other levels can be assigned by the system designer. Table 14.4 is a summary of interrupt sources and levels for MPC555 and according *ISR/PRIORITY* attribute value.

Table 14.4   Sources Versus Levels and PRIORITY for MPC555

Priority	Interrupt Source	PRIORITY
0(Highest)	IRQ0	15
1	Level0	14
2	IRQ1	13
3	Level 1	12
4	IRQ2	11
5	Level 2	10
6	IRQ3	9
7	Level 3	8
8	IRQ4	7
9	Level 4	6

Table 14.4   Sources Versus Levels and PRIORITY for MPC555

Priority	Interrupt Source	PRIORITY
10	IRQ5	5
11	Level 5	4
12	IRQ6	3
13	Level 6	2
14	IRQ7	1
15(Lowest)	Level 7	0
16-31	Reserved	

NOTE   The highest value 15 of the *ISR/PRIORITY* attribute accords to the highest interrupt level 0 and vice versa the lowest value 0 of the *ISR/PRIORITY* attribute accords to lowest interrupt level 15.

When an interrupt sharing vector offset 0xYYY00500 (external interrupt) occurs, its corresponding level is set in the interrupt request pending register. Each bit represents one level, hence up to 16 levels (or sources) are possible for MPC555. Individual interrupt levels are enabled and disabled by the interrupt mask register. Setting a bit enables a source of the corresponding level to cause an actual interrupt; clearing a bit disables the corresponding level.

Work Without Interrupt Dispatcher

If user does not want to use the interrupt dispatcher he has to use the "None" value for *InterruptDispatcher* attribute and in this case it may be useful to incorporate additional statements (connected with interrupt source determination) directly into MPC interrupt vector table prior to "ba <nameISR>" call. In this case, it is user's responsibility to save/restore affected registers between interrupt entry point and "ba <nameISR>" call. In this case the *PRIORITY* values for all ISRs are no significance but it is recommended make it like in cases with One and Multi Level interrupt dispatcher.

It is possible to work without *Interrupt Dispatcher* if only one interrupt (include *System/Second Timer*, but exclude DEC) is used in application**One Level Interrupt Dispatcher**

When an interrupt(s) occur Interrupt Dispatcher in the One Level mode calls the interrupt handler which corresponds to the interrupt

of highest level among pending interrupts. All interrupts are disabled during interrupt processing.

All ISR definition in the oil file shall have different values of the *PRIORITY* attribute in accordance with Table 14.4. ISR which is intended for the non external interrupt (DEC for instance) processing shall have the highest priority (16 for MPC555).

Multilevel Interrupt Dispatcher

OSEKturbo OS emulates multilevel interrupts by manipulating interrupt mask register bits. *Multilevel Interrupt Dispatcher* works as one level dispatcher but interrupts , so interrupts of higher levels are enabled After Interrupt processing ends the saved original interrupt mask is restored.

NOTE   During interrupt processing all interrupts of higher levels are enabled and all interrupts of same and lower levels are disabled. It is impossible to manipulate with separate bits of the interrupt mask when Multilevel Interrupt Dispatcher is used.

All ISR definition in the OIL file shall have different values of the *PRIORITY* attribute in accordance with Table 14.4. ISR which is intended for the non external interrupt (DEC for instance) processing shall have the highest priority for platform (16 for MPC555).

Stack Definition

MPC r1 register (stack pointer) should always contain word-aligned address, therefore all task and ISR stack sizes in OSEK OS application should be word-aligned (should be multiple of 4).

Recommendations on System Properties

FloatingPoint Property

It is recommended to turn OFF this property if the application does not use floating point arithmetic. It reduces the needed amount of RAM and increases OS performance. However, if there are floating point operations in the application then *FloatingPoint* shall be set to TRUE.

Timer Hardware

The special OS attributes are introduced to define hardware interrupt source and desired parameters for counter considered to be System (Second) Timer.

Note that counter assigned to System (Second) Timer uses interrupts from corresponding timer channel. At run time OSEKturbo OS enables CPU interrupts and timers interrupts corresponding to System and Second Timers. User shall not directly manipulate with System/Second timer hardware when OSEK is running. However the timers hardware may be initialized prior to calling to StartOS in main function. In this case the *Prescaler* attribute may be set to *USER* thus disabling timer prescaler reinitialization at OSEK startup. If *Prescaler* is set to *OS*, its *Value* is written to the prescaler bits of the correspondent timer. Note that the *Prescaler/Value* attribute value is not equal to divide factor of timer hardware.

The following statement should be used for system timer definition:

```
OS <name> {

...

    SysTimer = <SWCOUNTER / HWCOUNTER>{
      COUNTER = <CounterName>;
      ISRPRIORITY = <Priority>;
      TimerHardware = <HardwareType> {
        Prescaler = <PrescalerMode> {
          Value = <HardwarePrescaler>;
        };
        TimerModuloValue = <HardwareModulo>;
        Freeze = <TRUE / FALSE>;
        RTSEL = <OSCM / EXTCLK>;
      };
    };

...

};
```

This OSEK OS version contains system timer code developed for MPC555/56x target.The following hardware sources can be used for System and/or Second Timers:

- PIT (Periodic Interrupt Timer);
- DEC (Decrementer Counter);
- RTC (Real-Time Counter);
- TB0, TB1 - Time Base Reference registers 0,1 - used only for hardware counters.
-

PIT-based System(Second) Timer

The *ISRPRIORITY* attribute specifies the periodic interrupt request level. This attribute shall have one of the following values 0, 2, 4, 6, 8, 10, 12, 14. Value 0 accords to interrupt request level 7, value 14 accords to interrupt request level 0. For more detail information about the *ISRPRIORITY* values and Interrupt levels see ("External Interrupts Processing Background").

The *Prescaler/Value* attribute specifies the value of RTDIV (RTC and PIT clock divider and input source select) bits of the SCCR (System Clock Control Register). The value of this attribute equal 0 accords to PIT clock divided by four. The value equal to 1 accords to PIT clock divided by 256.

The *TimerModuloValue* attribute provides value of Periodic Interrupt Timer Count (PITC register) and is used to determine periodic interrupt rate.

The *Freeze* attribute specifies timer freeze bit (PITF) value of the Periodic Interrupt Status and Control Register (PISCR). If this attribute equal TRUE it means that PITF is set and the PIT stops while FREEZE is asserted. It may be used during the application debugging: if this attribute has value TRUE, timer will be frozen during interrupt processing in the debug mode.

The *RTSEL* attribute specifies clock source for PIT timer (SCCR/ RTSEL bit). The EXTCLK value means that external clock will be used as clock source, the OSCM value means that the main crystal oscillator will be used as clock source. This attribute shall have same value for System and Second timers if they are PIT and RTC.

RTC-based System(Second) Timer

The *ISRPRIORITY* attribute specifies the periodic interrupt request level. This attribute shall have one of the following values 0, 2, 4, 6, 8, 10, 12, 14. Value 0 accords to interrupt request level 7, value 14 accords to interrupt request level 0. For more detail information about the *ISRPRIORITY* values and Interrupt levels see ("External Interrupts Processing Background").

The *Prescaler/Value* attribute specifies the value of RTC and PIT clock divider and input source select (RTDIV) bit of the System Clock Control Register (SCCR) and Real-time clock freeze (4M) bit of the Real-Time Clock Status and Control Register (RTCSC). This attribute may have following values:

- 0 - both bits are cleared. It means that RTC and PIT clock divided by four and RTC assumes that it is driven by 20 MHz to generate the seconds pulse.
- 1 - the RTCSC/4M bit is cleared, SCCR/RTDIV bit is set. It means that RTC and PIT clock divided by 256 and RTC assumes that it is driven by 20 MHz.
- 2 - the RTCSC/4M bit is set, SCCR/RTDIV bit is cleared. It means that RTC and PIT clock divided by four and RTC assumes that it is driven by 4 MHz.
- 3 - both bits are set. It means that RTC and PIT clock divided by 256 and RTC assumes that it is driven by 4 MHz to generate the seconds pulse.

The *TimerModuloValue* attribute provides value of Real Time Clock Alarm Register (RTCAL) and is used to determine periodic interrupt rate.

The *Freeze* attribute specifies timer freeze bit (RTF) value of the Real-Time Clock Status and Control Register (RTCSCR). If this attribute equal TRUE it means that RTF is set and the RTC stops while FREEZE is asserted. It may be used during the application debugging: If this attribute has value TRUE, timer will be frozen during interrupt processing in the debug mode.

The *RTSEL* attribute specifies clock source for RTC timer (SCCR/RTSEL bit). The EXTCLK value means that external clock will be used as clock source, the OSCM value means that the main crystal

oscillator will be used as clock source. This attribute shall have same value for System and Second timers if they are PIT and RTC.

Please see MPC555 User's Manual for MPC555 timers configuration.

DEC-based System(Second) Timer

The *ISRPRIORITY* attribute specifies the periodic interrupt request level. This attribute shall have the 16 value

The *Prescaler/Value* attribute specifies the value of TBS (Time Base Source) bit of the SCCR (System Clock Control Register). If the value of this attribute is 0 then OSCCLK clock divided by 4 or 16 depending on value of *TargetMCU/ClockMultiplier.* The value of 1 specifies system clock divided by 16 as a clock source. Please see the hardware manual for more details.

The *TimerModuloValue* attribute provides value to be (re)loaded into the decrementer register (DEC) and is used to determine decrementer interrupt rate.

The *Freeze* attribute specifies Time Base Freeze bit (TBF) value of the Time Base Control and Status Register (TBSCR). If this attribute equal TRUE the time base and decrementer stop while FREEZE is asserted. It may be used during the application debugging: If this attribute has value TRUE, timer will be frozen during processing of breakpoint in the debugger.

System and Second Timers

If both System and Second timers are used in the application and DEC timer is not used (DEC uses separate vector) then the interrupt dispatcher shall be used, otherwise it is user's responsibility to define which external interrupt occurred and call appropriated interrupt handler including OSISRSystemTimer and OSISRSecondTimer roots.

Example of Timers Configuration

For example to configure a system timer as a Hardware counter with 1000ns tick duration and the 1ms (1kHz) second timer as a Software counter on an MPC555 with 4MHz crystal and 40MHz CPU frequency the following attributes may be set in the OIL file:
```
OS OsName {
```

```
...
    ClockFrequency = 4000; // oscillator frequency
    ClockMultiplier = 10;
    ClockDivider = 1;
    SysTimer = HWCOUNTER {
      COUNTER = CounterName1;
      ISRPRIORITY = 10;
      TimerHardware = TIMTB0 {
        Period = 1000;
        Prescaler = OS {
          Value = AUTO;
        };
      };
    };
...
    SecondTimer = SWCOUNTER{
      COUNTER = CounterName2;
      ISRPRIORITY = 0;
      TimerHardware = PIT{
        Prescaler = OS {
          Value = 0;  // no divide, 4 MHz
        };
        TimerModuloValue = 999;
      };
    };
...
};
```

For the system hardware timer value of 1000ns for *Period* causes
SysGen to calculate 0 (divide by 4) as a *Value* for *Prescaler,* resulting
source frequency for TimeBase (TMBCLK) is 1MHz.

For the *SecondTimer Prescaler Value* = 0 (factor 4) divides the
oscillator clock to 1MHz, and the *TimerModuloValue* = 999 causes an
interrupt to occur every 1000 ticks = 1ms.

MPC565/563/561 Features

The enhanced interrupt controller of MPC565/561 is supported in
addition to all MPC features. The above "MPC555 Features" is fully

applicable for MPC565/563/561. Code compression is not supported for MPC565/563/561.

Regular Interrupt Controller Operation (MPC555 Compatible Mode)

The above "Interrupt Dispatcher" is fully applicable for this mode of the interrupt controller. In this mode *InterruptDispatcher* attribute shall have the value of *None, OneLevel* or *Multilevel.*

Enhanced Interrupt Controller

The enhanced interrupt controller operation may be turned on for MPC565/563/561 by assigning an "*Enhanced*" value to *InterruptDispatcher* attribute. In this mode the 32 IMB interrupt levels will be latched by USIU using eight IMB interrupt lines and two lines of ILBS via time multiplexing scheme defined by the UIMB module. In addition to the IMB interrupt sources the external interrupts and timer interrupts are available by the same way as in the regular scheme.

NOTE   Current version of OSEKturbo OS/MPC5xx does not support the time multiplexing scheme. User should choice the multiplexing scheme by the UIMB module configuration.

Table 14.5 is a summary of interrupt sources and levels for MPC565/563/561 and according *ISR/PRIORITY* attribute value.

For system operating properly all ISR definition in the OIL file shall have different values of the *PRIORITY* attribute in accordance with Table 14.5. ISR which is intended for the non external interrupt processing (DEC for instance) shall have the highest priority for platform (48 for MPC565 with Enhanced Interrupt Controller turned on).

When Enhanced Interrupt Controller is turned on the user ISRs of category 1 are invoked directly by hardware without any interaction with OS. It is the user responsibility to enable interrupts inside ISR's code via setting EE bit in MSR in order to use nesting of ISRs of category 1.

Table 14.5   Sources Versus Levels and PRIORITY for MPC565/563/561

HW Priority	Interrupt Source	OS PRIORITY
0(Highest)	EXT_IRQ 0	47
1	Level 0	46
2	IMB_IRQ 0	45
3	IMB_IRQ 1	44
4	IMB_IRQ 2	43
5	IMB_IRQ 3	42
6	EXT_IRQ 1	41
7	Level 1	40
8	IMB_IRQ 4	39
9	IMB_IRQ 5	38
10	IMB_IRQ 6	37
11	IMB_IRQ 7	36
12	EXT_IRQ 2	35
13	Level 2	34
14	IMB_IRQ 8	33
15	IMB_IRQ 9	32
16	IMB_IRQ 10	31
17	IMB_IRQ 11	30
18	EXT_IRQ 3	29
19	Level 3	28
20	IMB_IRQ 12	27
21	IMB_IRQ 13	26
22	IMB_IRQ 14	25
23	IMB_IRQ 15	24
24	EXT_IRQ 4	23
25	Level 4	22
26	IMB_IRQ 16	21
27	IMB_IRQ 17	20
28	IMB_IRQ 18	19
29	IMB_IRQ 19	18
30	EXT_IRQ 5	17

Table 14.5   Sources Versus Levels and PRIORITY for MPC565/563/561

HW Priority	Interrupt Source	OS PRIORITY
31	Level 5	16
32	IMB_IRQ 20	15
33	IMB_IRQ 21	14
34	IMB_IRQ 22	13
35	IMB_IRQ 23	12
36	EXT_IRQ 6	11
37	Level 6	10
38	IMB_IRQ 24	9
39	IMB_IRQ 25	8
40	IMB_IRQ 26	7
41	IMB_IRQ 27	6
42	EXT_IRQ 7	5
43	Level 7	4
44	IMB_IRQ 28	3
45	IMB_IRQ 29	2
46	IMB_IRQ 30	1
47 (Lowest)	IMB_IRQ 31	0

To have ability to use the Enhanced Interrupt Controller it is necessary to locate the "extTable" section on the address no less than 0x1d000 aligned to 0x800 boundary. See the makefiles in the `sample\standard\mpc565\msmak` directory for additional information.

DEC-based System(Second) Timer

The *ISRPRIORITY* attribute specifies the periodic interrupt request level. This attribute shall have the 48 value if the *InterruptDispatcher* value is "Enhanced".

15

Application Troubleshooting

In this chapter some advice is given which may be useful for developers working with the OSEK Operating System.

This chapter consists of the following sections:

- System Generation
- Stack Errors
- Unregistered External Interrupts
- Known Problems

System Generation

The System Generator is used to generate the code for the OSEK Operating System kernel and all application objects (tasks, messages, etc.). This tool processed the configuration file created by the user and reports about inconsistencies and errors in it. Most of possible mistakes in application configuration process can be eliminated with the help of SG. See "System Configuration" and "Building of Application" about system generation process.

If an undocumented problem arises please provide us with the detailed description of it and we will help to resolve the problem. See "Technical Support Information" for contact information.

Stack Errors

Stack errors may be due to the stack pointer being incorrect or to stack content being corrupted. Stack content problems are possible if pointers are used to access stack variables, but stack pointer problems seem to be more common. The symptom of either problem is usually a task or ISR executing normally, but then when

For More Information: www.freescale.com

a return is performed, the program executes at some incorrect address.

Tasks should have enough stack for their execution, therefore it is recommended to pay attention on task definition statements to provide each task with a needed amount of stack. See "PowerPC Platform-Specific Features".

NOTE The optional stack overflow checking during run time are implemented in OSEKturbo OS for debugging application. See "Stack Overflow Checking".

Unregistered External Interrupts

If interrupt dispatcher is used then external interrupts shall be registered in the system that is they shall be defined in the application oil file. System will stop on the 0xYYY1000 vector if unregistered interrupt occurs. OSEK OS does not support reset of all interrupt sources during initialization.

Known Problems

Troubleshooting

Problem A: Error while running SETUP.EXE from a network drive.

Resolving: Start `SETUP.EXE` from local drive.

Problem B: Installation program produces error message: 'Cannot detect installation media. Installation failed'.

Reason: Installation files were copied into NTFS partition, 'archive' file attribute flags are cleared for some of them and 'compress' file attribute flags are set for some of them.

Workaround: Set 'archive' file attribute flag for all the installation files.

Problem C: Development software does not work under MS Windows.

Reason 1: Inappropriate development hardware or software is used.

Reason 2: OSEK OS/mpc software is installed into the directory with spaces (like "`C:\Program Files`").

Workaround 1: Install the software into the directory without spaces (for example: "C:\metrowerks\osek").

Reason 3: Environment variables `OSEKDIR, DIABDIR, CWDIR` are not correct.

*Workaround 1:*Set correct environment variables.

Problem D: The NMAKE or the MAKE generates a wrong output.

Reason 1: The needed directories and files are not installed properly.

Resolving 1: Check the integrity and consistency of the package using `FILELIST.TXT`.

Reason 2: The needed software is not installed or installed improperly.

Resolving 1: Check the installed software (listed above).

Resolving 2: Check the correctness of the software installation (environment variables, etc.)

Problem E: Shared files are not removed from `BIN` directory during uninstallation of OSEK OS.

Reason: Current version of uninstall program (file `IsUninst.exe` ver. 5.10.145.0 from Windows NT Service Pack 4) processes shared files with error.

Resolving: Get new version of `IsUninst.exe` program.

Workaround: Remove shared files manually after uninstallation.

Problem F: Error "The parameter is incorrect" during installation.

Reason: Too long full name (include parent directories) of the directory with installation media.

Workaround: Move installation media to directory with shorter full path.

Problem G: Some icon does not created in Start menu after installation.

Reason: There is not enough space on target drive.

Workaround: Reinstall OSEK/OS to other drive.

16

System Services

This chapter provides a detailed description for all OSEK Operating System run-time services, with appropriate examples.

This chapter consists of the following sections:

- General
- Task Management Services
- ISR Management Services
- Resource Management Services
- Event Management Services
- Counter Management Services
- Alarm Management Services
- Communication Management Services
- Debugging Services
- Operating System Execution Control

General

This chapter provides detailed description of all OSEK OS run-time services including hook routines. Also declarations of system objects – the constructional elements – are described here. The services are arranged in logical groups – for the task management, the interrupt management, etc.

Examples of code are also provided for every logical group. These examples have no practical meaning, they only show how it is possible to use OS calls in an application.

The following scheme is used for service description:

Declaration element:

Syntax: Operating System interface in ANSI-C syntax.

Input: List of all input parameters.

Description: Explanation of the constructional element.

Particularities: Explanation of restrictions relating to the utilization.

Conformance: Specifies the Conformance Classes where the declaration element is provided.

Service description:

Syntax: Operating System interface in ANSI-C syntax.

Input: List of all input parameters.

Output: List of all output parameters. Transfers via the memory use the memory reference as input parameter and the memory contents as output parameter. To clarify the description, the reference is already specified among the output parameters.

Description: Explanation of the functionality of the operating system service.

Particularities: Explanations of restrictions relating to the utilization of the service.

Status: List of possible return values if service returns status of *StatusType* type.

- Standard:

  List of return values provided in the operating system's standard version. Special case – service does not return status.

- Extended:

  List of additional return values in the operating system's extended version.

Conformance: Specifies the Conformance Classes where the service is provided.

Task Management Services

Data Types

The OSEK OS establishes the following data types for the task management:

- *TaskType* – the abstract data type for task identification

- *TaskRefType* – the data type to refer variables of the *TaskType* data type. Reference or pointer to *TaskType* variable can be used instead of *TaskRefType* variable

- *TaskStateType* – the data type for variables to store the state of a task

- *TaskStateRefType* – the data type to refer variables of the *TaskStateType* data type. Reference or pointer to *TaskStateType* variable can be used instead of *TaskStateRefType* variable

Constants

The following constants are used within the OSEK Operating System to indicate task states:

- *RUNNING* – constant of data type *TaskStateType* for task state *running*

- *WAITING* – constant of data type *TaskStateType* for task state *waiting*

- *READY* – constant of data type *TaskStateType* for task state *ready*

- *SUSPENDED* – constant of data type *TaskStateType* for task state *suspended*

The following constant is used within the OSEK OS to indicate task:

- *INVALID_TASK* – constant of data type *TaskType* for undefined task

Conventions

Within the application of the OSEK OS a task should be defined according to the following pattern:

```
TASK ( <name of task> )
{
...
}
```

The name of the task function will be generated from *<name of task>* by macro *TASK*.

Task Declaration

The constructional statement *DeclareTask* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax: `DeclareTask( <name of task> );`

Input: *<name of task>* – a reference to the task.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are defined at system generation phase.

Conformance: BCC1, ECC1

ActivateTask

Syntax: `StatusType ActivateTask( TaskType <TaskID> );`

Input: *<TaskID>* – a reference to the task.

Output: None.

Description: The specified task *<TaskID>* is transferred from the *suspended* state into the *ready* state.

Particularities: The service may be called both on the task level (from a task) and the interrupt level (from ISR).

In the case of calling from ISR, the operating system will reschedule tasks only after the ISR completion.

Status:
- Standard:
    - E_OK – no error.
    - E_OS_LIMIT – too many task activations of the specified task.
- Extended:
    - E_OS_ID – the task identifier is invalid.

Conformance: BCC1, ECC1

TerminateTask

Syntax: `StatusType TerminateTask( void );`

Input: None.

Output: None.

Description: This service causes the termination of the calling task. The calling task is transferred from the *running* state into the *suspended* state.

Particularities: The resources occupied by the task shall be released before the call to *TerminateTask* service. If the call was successful, *TerminateTask* does not return to the call level and enforces a rescheduling. Ending a task function without calling *TerminateTask* or *ChainTask* service is strictly forbidden.

If the system with extended status is used, the service returns in case of error, and provides a status which can be evaluated in the application.

There are following limitations for BCC1 class if *FastTerminate* is set to TRUE: *TerminateTask* service shall be called in task function body from the function level; in STANDARD status this service does not return a status and can not be used in expressions.

The service call is allowed on task level only.

Status:
- Standard:
  - No return to call level.
- Extended:
  - E_OS_RESOURCE – the task still occupies resources.
  - E_OS_CALLEVEL – a call at the interrupt level.

Conformance: BCC1, ECC1

ChainTask

Syntax: `StatusType ChainTask( TaskType <TaskID> );`

Input: *<TaskID>* – a reference to the sequential succeeding task to be activated.

Output: None.

Description: This service causes the termination of the calling task. After termination of the calling task a succeeding task *<TaskID>* is transferred from the *suspended* state into the *ready* state. Using this service, it ensures that the succeeding task only starts to run after the calling task has been terminated.

Particularities: The resources occupied by the calling task shall be released before the call to *ChainTask* service. If the call was successful, *ChainTask* does not return to the call level and enforces a rescheduling. Ending a task function without calling *TerminateTask* or *ChainTask* service is strictly forbidden.

If the succeeding task is identical with the current task, this does not result in multiple requests.

The service returns in case of error and provides a status which can be evaluated by the application.

There are following limitations for BCC1 class if *FastTerminate* is set to TRUE: *ChainTask* service shall be called in task function body from the function level; in STANDARD status this service does not return a status and can not be used in expressions.

The service call is allowed on task level only.

Status: • Standard:
  – No return to call level.
  – E_OS_LIMIT – too many activations of *<TaskID>*.
• Extended:
  – E_OS_ID – the task identifier is invalid.
  – E_OS_RESOURCE – the calling task still occupies resources.
  – E_OS_CALLEVEL – a call at the interrupt level.

Conformance: BCC1, ECC1

Schedule

Syntax: `StatusType Schedule( void );`

Input: None.

Output: None.

Description: If there is a task in *ready* state with priority higher than assigned priority of calling task, the internal resource (if any) of the task is released, the calling task is put into the *ready* state and the higher-priority task is transferred into the *running* state. Otherwise the calling task is continued.

Particularities: Rescheduling can only take place if an internal resource is assigned to the calling task during system generation (non-preemtable tasks are considered as a tasks with internal resource of highest priority). For these tasks, Schedule enables a processor assignment to other tasks with assigned priority not higher than the ceiling priority of the internal resource and higher than the assigned priority of the calling task. When returning from Schedule, the internal resource is taken again. This service has no influence on tasks with no internal resource assigned (preemptable tasks).

The service call is allowed on task level only.

Status: • Standard:

– E_OK – no error.

• Extended:

– E_OS_CALLEVEL – a call at the interrupt level.

– E_OS_RESOURCE - calling task occupies resources.

Conformance: BCC1, ECC1

GetTaskId

Syntax: `StatusType GetTaskID( TaskRefType <TaskIDRef> );`

Input: None.

Output: *<TaskIDRef>* – a pointer to the variable contained reference to the task which is currently running. The service saves the task reference into the variable, that is addressed by pointer *<TaskIDRef>*. Reference to *TaskType* variable can be used instead of *TaskRefType* variable.

Description: This service returns reference to the task which is currently running. If no task currently running, the service returns INVALID_TASK constant.

Particularities: The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

Status: • Standard:

– E_OK – no error.

• Extended:

– None.

Conformance: BCC1, ECC1

GetTaskState

Syntax:
```
StatusType GetTaskState( TaskType <TaskID>,
                 TaskStateRefType <StateRef> );
```

Input: *<TaskID>* – a reference to the task.

Output: *<StateRef>* – a pointer to the state of task. The service saves the task state into the variable, that is addressed by pointer *<StateRef>*. Reference to *TaskStateType* variable can be used instead of *TaskStateRefType* variable.

Description: The service returns the state of the specified task *<TaskID>* (*running*, *ready*, *waiting*, *suspended*) at the time of calling *GetTaskState*.

Particularities: The service may be called both on the task level (from a task) and the interrupt level (from ISR). This service may be called from *ErrorHook*, *PreTaskHook*, *PostTaskHook* hook routines.

Within a full-preemptive system, calling this operating system service only provides a meaningful result if the task runs in an interrupt disabling state at the time of calling. When a call is made from a task in a full-preemptive system, the result may already be incorrect at the time of evaluation.

When the service is called for a task, which is multiply activated, the state is set to *running* if any instance of the task is running.

Status: • Standard:

– E_OK – no error.

• Extended:

– E_OS_ID – the task identifier is invalid.

Conformance: BCC1, ECC1

Examples for Task Management Services

The example below shows use of all OSEK OS task management services. PreTaskHook is used to count switches to tasks. Three tasks are declared in the OIL configuration file:

```
...
  OS os1{
    ...
    PRETASKHOOK = TRUE;
  };
...
TASK TaskA {
  PRIORITY = 3;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
};
TASK TaskB {
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = EventTaskB;
};
TASK TaskC {
  PRIORITY = 1;
  SCHEDULE = NON;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = EventTaskC;
};

...
```

The C-language example file:

```
int CntTaskA;
```

```
int CntTaskB;
int CntTaskC;

void PreTaskHook( void );
{
  GetTaskId( &task );

  switch( task )              /* increment corresponding counter */
  {
    case TaskA:
      CntTaskA++; break;
    case TaskB:
      CntTaskB++; break;
    case TaskC:
      CntTaskC++; break;
  }
}

TASK( TaskA )
{
  TaskType task;
  ...                /* any user's code */

  ActivateTask( TaskB ); /* activate TaskB */
  ...                /* any user's code */
  ChainTask( TaskC );
}

TASK( TaskB )
{
  TaskStateType state;
  ...                /* any user code */

  GetTaskState( TaskC, &state );  /* check the state of TaskC */
  switch( state )          /* and perform appropriate actions */
    {
      case READY:    break;
      case WAITING:  SetEvent( TaskC, EventTaskC );
            break;
      case SUSPENDED: ChainTask( TaskC );
            break;
    }
```

```
   ...                    /* any user code */
   TerminateTask();
}


TASK( TaskC )
{
   TaskStateType stateA, stateB;
   ...                    /* any user code */

   while( 1 )
   {
     WaitEvent( EventTaskC );
     GetTaskState( TaskA, &stateA );
     GetTaskState( TaskB, &stateB );
     if( stateA == READY && stateB == SUSPENDED )
       ChainTask( TaskB );
     if( stateB == READY && stateA == SUSPENDED )
       ChainTask( TaskA );
     if( stateA == READY && stateB == READY )
       Schedule();   /* yield CPU to a higher-priority task */
     ...              /* any user code */
   }
}
```

ISR Management Services

Data Types

No special data types are defined for the OSEK interrupt handling
functionality.

Conventions

Within the application an Interrupt Service Routine should be
defined according to the following pattern:

```
ISR( <name of ISR>)
{
```

For More Information: www.freescale.com

```
...
}
```

The keyword *ISR* is the macro for OS and compiler specific interrupt function modifier, which is used to generate valid code to enter and exit ISR.

ISR Declaration

The constructional statement *DeclareISR*[1] may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax: `DeclareISR( <name of ISR> );`

Input: `<name of ISR>` – a reference to the ISR.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are defined at system generation phase.

Conformance: BCC1, ECC1

EnableAllInterrupts

Syntax: `void EnableAllInterrupts ( void );`

Input: None.

Output: None.

Description: This service restores the interrupts state saved by *DisableAllInterrupts* service. It can be called after *DisableAllInterrupts* only. This service is a counterpart of *DisableAllInterrupts* service, and its aim is the completion of the critical section of code. No API service calls are allowed within this critical section.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

---

[1] This declaration is not defined by *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

This service is intended to control only the "EE" bit in the Machine State Register (MSR).This service does not support nesting.

Status:
- Standard:
  - None.
- Extended:
  - None.

Conformance: BCC1, ECC1

DisableAllInterrupts

Syntax: `void DisableAllInterrupts ( void );`

Input: None.

Output: None.

Description: This service saves the current interrupts state and disables all hardware interrupts. This service is intended to start a critical section of the code. This section must be finished by calling the *EnableAllInterrupts* service. No API service calls are allowed within this critical section.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

This service is intended to control only the "EE" bit in the Machine State Register (MSR).

This service does not support nesting.

Status:
- Standard:
  - None.
- Extended:
  - None.

Conformance: BCC1, ECC1

ResumeAllInterrupts

Syntax: `void ResumeAllInterrupts ( void );`

Input:	None.
Output:	None.
Description:	This service restores the recognition status of all interrupts saved by *SuspendAllInterrupts* service.
Particularities:	The service may be called from an ISR category 1 and category 2, from the alarm-callbacks and from the task level, but not from hook routines.

Particularities: This service is the counterpart of the *SuspendAllInterrupts* service, which must have been called before, and its aim is the completion of the critical section of code. No API service calls beside *SuspendAllInterrupts/ResumeAllInterrupts* pairs and *SuspendOSInterrupts/ResumeOSInterrupts* pairs are allowed within this critical section.

SuspendAllInterrupts/ResumeAllInterrupts can be nested. In case of nesting pairs of the calls *SuspendAllInterrupts* and *ResumeAllInterrupts* the interrupt recognition status saved by the first call of *SuspendAllInterrupts* is restored by the last call of the *ResumeAllInterrupts* service.

If *ErrorHook* is defined and *STATUS* is set to EXTENDED this service calls[1] *ErrorHook* with *E_OS_SYS_ORDER* status in case of a wrong sequence of interrupt management functions calls. This check is limited to 32 levels of nesting pairs of Suspend/Resume functions.

Status:
- Standard:
  - None.
- Extended:
  - None.

Conformance: BCC1, ECC1

SuspendAllInterrupts

Syntax: `void SuspendAllInterrupts ( void );`

Input: None.

---

[1] this is OSEKturbo extension of OSEK OS

Output: None.

Description: This service saves the recognition status of all interrupts and disables all interrupts for which the hardware supports disabling.

Particularities: The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from hook routines.

This service is intended to protect a critical section of code from interruptions of any kind. This section must be finished by calling the *ResumeAllInterrupts* service. No API service calls beside *SuspendAllInterrupts/ResumeAllInterrupts* pairs and *SuspendOSInterrupts/ResumeOSInterrupts* pairs are allowed within this critical section.

If ResumeAllInterrupts service was not called after this service and before point of rescheduling then OSEKturbo dispatcher calls[1] ErrorHook, if it is defined, with parametr E_OS_SYS_ORDER.

Status: • Standard:

– None.

• Extended:

– None.

Conformance: BCC1, ECC1

ResumeOSInterrupts

Syntax: `void ResumeOSInterrupts ( void );`

Input: None.

Output: None.

Description: This service restores the interrupts state saved by *SuspendOSInterrupts* service. It can be called after *SuspendOSInterrupts* only. This service is the counterpart of *SuspendOSInterrupts* service, and its aim is the completion of the critical section of code. No API service calls beside *SuspendAllInterrupts/ResumeAllInterrupts* pairs and

---

[1.] this is OSEKturbo extension of OSEK OS

SuspendOSInterrupts/ResumeOSInterrupts pairs are allowed within this critical section.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

In case of nesting pairs of the calls *SuspendOSInterrupts* and *ResumeOSInterrupts* the interrupt recognition status saved by the first call of *SuspendOSInterrupts* is restored by the last call of the *ResumeOSInterrupts* service.

If *ErrorHook* is defined and *STATUS* is set to EXTENDED this service calls[1] *ErrorHook* with *E_OS_SYS_ORDER* status in case of a wrong sequence of interrupt management functions calls. This check is limited to 32 levels of nesting pairs of Suspend/Resume functions.

If no ISRs of category 2 are defined, then this service does nothing.

Status: • Standard:

– None.

• Extended:

– None.

Conformance: BCC1, ECC1

SuspendOSInterrupts

Syntax: `void SuspendOSInterrupts ( void );`

Input: None.

Output: None.

Description: This service saves current interrupt state and disables all interrupts of category 2. Interrupts category 1 which priority is not higher than priority of any ISR category 2 are disabled also. This service is intended to start a critical section of the code. This section must be finished by calling the *ResumeOSInterrupts* service. No API service calls beside *SuspendAllInterrupts/ResumeAllInterrupts* pairs and

---

[1] this is OSEKturbo extension of OSEK OS

SuspendOSInterrupts/ResumeOSInterrupts pairs are allowed within this critical section.

Particularities: The service may be called from an ISR and from the task level, but not from hook routines.

In case of nesting pairs of the calls *SuspendOSInterrupts* and *ResumeOSInterrupts* the interrupt status saved by the first call of *SuspendOSInterrupts* is restored by the last call of the *ResumeOSInterrupts* service.

If ResumeOSInterrupts service was not called after this service and before point of rescheduling then OSEKturbo dispatcher calls[1] ErrorHook, if it is defined, with parametr E_OS_SYS_ORDER.

If no ISRs of category 2 are defined, then this service does nothing.

Status: • Standard:
  – None.
  • Extended:
  – None.

Conformance: BCC1, ECC1

Examples for Interrupt Management Services

Below examples for ISR category 1 and 2 are presented.

The following definitions can be made in the definition file:

```
...
OS myOS {
  ...
  IsrStackSize = 256;
  MessageCopyAllocation = OS;
  ...
};
TASK TaskB {
  PRIORITY = 2;
  SCHEDULE = FULL;
```

[1.] this is OSEKturbo extension of OSEK OS

For More Information: www.freescale.com

```
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = EventTaskB;
};
TASK IndTask {
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
};
COUNTER Ctr1 {
  MINCYCLE = 1;
  MAXALLOWEDVALUE = 24;
  TICKSPERBASE = 1;
};
MESSAGE Temp {
  TYPE = UNQUEUED;
  CDATATYPE = "char";
  ACTION = NONE;
};
MESSAGE Wrn {
  TYPE = UNQUEUED;
  CDATATYPE = "MSGCTYPE";
  ACTION = NONE;
};
ISR ISR1_handler {
  CATEGORY = 1;
  PRIORITY = 8;
};
ISR ISR2_handler {
  CATEGORY = 2;
  PRIORITY = 6;
};
...
```

The C-language code can be the following:

```
char CREG, DREG;
char data1;
...
```

```
/* ISR category 1: */
ISR( ISR1_handler )
{
  if( CREG != 0xC0 ) CREG |= 0x40;
  else CREG |= 0x03;
  DREG = data1;
}
TaskStateType stateB;
...
/* ISR category 2: */
ISR( ISR2_handler )
{
  CounterTrigger( Ctr1 );
  GetTaskState( TaskB, &stateB );
  if( stateB == SUSPENDED ) ActivateTask( TaskB );
}
```

Resource Management Services

Data Types

The OSEK OS establishes the following data type for the resource management:

- *ResourceType* – the abstract data type for referencing a resource

The only data type must be used for operations with resources.

Constants

- *RES_SCHEDULER* – constant of data type *ResourceType* corresponded to Scheduler Resource (see "Scheduler as a Resource")

Resource Declaration

The declaration statement *DeclareResource* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax: `DeclareResource( <name of resource> );`

Input: *<name of resource>* – a reference to the resource.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are defined at system generation phase.

Conformance: BCC1, ECC1

GetResource

Syntax: `StatusType GetResource( ResourceType <ResID> );`

Input: *<ResID>* – a reference to the resource.

Output: None.

Description: This service changes current priority of the calling task or ISR according to ceiling priority protocol for resource management. *GetResource* serves to enter critical section in the code and blocks execution of any task or ISR which can get the resource *<ResID>*. A critical section must always be left using *ReleaseResource* within the same task or ISR.

Particularities: This function is fully supported in all Conformance Classes. It is OSEKturbo extension of OSEK OS because OSEK/VDX specifies full support only beginning from BCC2.

Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden.

The service call is allowed on task level and ISR level, but not in hook routines.

This service is not implemented if no *standard* resources are defined in the configuration file. The code supporting scheduler resource is excluded from application, if `ResourceScheduler` is defined as `FALSE` in OIL.

Regarding Extended Tasks, please note that *WaitEvent* within a critical section is prohibited.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the resource identifier is invalid.
  - E_OS_ACCESS – attempt to get resource which is already occupied by any task or ISR, or the assigned in OIL priority of the calling task or interrupt routine is higher than the calculated ceiling priority.

Conformance: BCC1, ECC1

ReleaseResource

Syntax: `StatusType ReleaseResource( ResourceType <ResID> );`

Input: *<ResID>* – a reference to the resource.

Output: None.

Description: This call serves to leave the critical sections in the code that are assigned to the resources referenced by *<ResID>*. A *ReleaseResource* call is a counterpart of a *GetResource* service call. This service returns task or ISR priority to the level saved by corresponded *GetResource* service.

Particularities: This function is fully supported in all Conformance Classes. It is OSEKturbo extension of OSEK OS because OSEK/VDX specifies full support only begining from BCC2.

Nested resource occupation is only allowed if the inner critical sections are completely executed within the surrounding critical section. Nested occupation of one and the same resource is forbidden.

The service call is allowed on task level and ISR level, but not in hook routines.

This service is not implemented if no *standard* resources are defined in the configuration file. The code supporting scheduler resource is excluded from application, if scheduler resource is not defined on OIL.

Status:
- Standard:

- – E_OK – no error.
- • Extended:
  - – E_OS_ID – the resource identifier is invalid.
  - – E_OS_NOFUNC – attempt to release a resource which is not occupied by any task or ISR, or another resource has to be released before.
  - – E_OS_ACCESS – attempt to release a resource which has a lower ceiling priority than the assigned in OIL priority of the calling task or interrupt routine. This error code returned only if E_OS_NOFUNC was not returned.

Conformance: BCC1, ECC1

Examples of Using Resources

The example below presents resource management directives.

The following definitions can be made in the definition file:

```
...
TASK TaskA {
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = EventTaskA;
  RESOURCE = SCI_res;
  RESOURCE = TASKB_res;
};
TASK TaskB {
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  RESOURCE = TASKB_res;
};
TASK TaskC {
  PRIORITY = 3;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
```

```
};
ISR SCI_handler {
  PRIORITY = 4;
  CATEGORY = 2;
  RESOURCE = SCI_res;
};

RESOURCE SCI_res {
      RESOURCEPROPERTY = STANDARD;
};
RESOURCE TASKB_res {
      RESOURCEPROPERTY = STANDARD;
};
...
```

The C-language code can be the following:

```
TASK( TaskA )
{
...     /* user's code */
        /* occupy the SCI resource to disable SCI_handler */
GetResource( SCI_res );
...     /* user's code which configures SCI */
        /* release the SCI resource to enable SCI_handler */
ReleaseResource( SCI_res );
...
        /* occupy the resource to avoid starting TaskB */
GetResource( TASKB_res );
ActivateTask( TaskB );
        /* occupy the scheduler to disable rescheduling */
GetResource( RES_SCHEDULER );
ActivateTask( TaskC );
...     /* user's code */
        /* release the scheduler resource */
ReleaseResource( RES_SCHEDULER );
        /* TaskC is started here */
...     /* user's code */
        /* release the TaskB resource */
ReleaseResource( TASKB_res );
        /* TaskB is started here */
...     /* user's code */
TerminateTask();
```

For More Information: www.freescale.com

```
}

TASK( TaskB )
{
...      /* user's code */
TerminateTask();
}

TASK( TaskC )
{
...      /* user's code */
TerminateTask();
}

ISR( SCI_handler )
{
...      /* user's code */
...      /* ISR can be disabled using a resource */
}
```

Event Management Services

Data Types

The OSEK Operating System establishes the following data types
for the event management:

- *EventMaskType* – the data type of the event mask
- *EventMaskRefType* – the data type of the pointer to an event mask

The only data types must be used for operations with events.

Event Declaration

The declaration statement *DeclareEvent* may be used for
compatibility with previous OSEK versions. It may be omitted in
application code.

Syntax: `DeclareEvent( <name of event> );`

Input: *<name of event>* – event name.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are defined at system generation phase.

Conformance: ECC1

SetEvent

Syntax:
```
StatusType SetEvent( TaskType <TaskID>,
            EventMaskType <Mask> );
```

Input: *<TaskID>* – a reference to the task for which one or several events are to be set.

<Mask> – an event mask to be set.

Output: None.

Description: This service is used to set one or several events of the desired task according to the event mask. If the task was *waiting* for at least one of the specified events, then it is transferred into the *ready* state. The events not specified by the mask remain unchanged. Only an extended task which is not suspended may be referenced to set an event.

Particularities: It is possible to set events for the running task (task-caller).

The service call is allowed on task level and ISR level, but not in hook routines.

This service is not implemented if no events are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the task identifier is invalid.
  - E_OS_ACCESS – the referenced task is not an Extended Task.
  - E_OS_STATE – the referenced task is in the suspended state.

Conformance: ECC1

ClearEvent

Syntax: `StatusType ClearEvent( EventMaskType <Mask> );`

Input: *<Mask>* – an event mask to be cleared.

Output: None.

Description: The task which calls this service defines the event which has to be cleared.

Particularities: The system service *ClearEvent* can be called from extended tasks which own an event only.

This service is not implemented if no events are defined in the configuration file.

Status:
- Standard:
  – E_OK – no error.
- Extended:
  – E_OS_ACCESS – the calling task is not an Extended Task.
  – E_OS_CALLEVEL – a call at the interrupt level is not allowed.

Conformance: ECC1

GetEvent

Syntax: `StatusType GetEvent( TaskType <TaskID>, EventMaskRefType <Event> );`

Input: *<TaskID>* – a reference to the task whose event mask is to be returned.

Output: *<Event>* – a pointer to the variable of the return state of events.

Description: The event mask which is referenced to in the call is filled according to the state of the events of the desired task. Current state of events is returned but not the mask of events that task is waiting for.

It is possible to get event mask of the running task (task-caller).

Particularities: The referenced task must be an extended task and it can not be in *suspended* state.

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

This service is not implemented if no events are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the task identifier is invalid.
  - E_OS_ACCESS – the referenced task is not an Extended Task.
  - E_OS_STATE – the referenced task is in the suspended state.

Conformance: ECC1

WaitEvent

Syntax: `StatusType WaitEvent( EventMaskType <Mask> );`

Input: *<Mask>* – an event mask to wait for.

Output: None.

Description: The calling task is transferred into the *waiting* state until at least one of the events specified by the mask is set. The task is kept the *running* state if any of the specified events is set at the time of the service call.

Particularities: This call enforces the rescheduling, if the wait condition occurs.

All resources occupied by the task must be released before *WaitEvent* service call.

The service can be called from extended tasks which own an event only.

This service is not implemented if no events are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.

- Extended:

  – E_OS_ACCESS – the calling task is not an Extended Task.

  – E_OS_RESOURCE – the calling task occupies resources.

  – E_OS_CALLEVEL – a call at the interrupt level is not allowed.

Conformance: ECC1

Examples of Using Events

The example below shows how events can be used in the OSEK Operating System.

The following definitions can be made in the definition file:

```
...
TASK TASK_A {
  PRIORITY = 3;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = ExtEvent1;
  EVENT = ExtEvent2;
  EVENT = XEvent;
  EVENT = YEvent;
  EVENT = Z1_FLG;
  EVENT = Z2_FLG;
};
TASK TASK_B {
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = DgrAlmEvent;
  EVENT = ExtEvent2;
};
TASK TASK_C {
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
```

```
    ACTIVATION = 1;
};
COUNTER DgrCnt {
  MAXALLOWEDVALUE = 150;
  TICKSPERBASE = 1;
  MINCYCLE = 1;
};
ALARM AWAKE {
  COUNTER = DgrCnt;
  ACTION = SETEVENT {
    TASK = TASK_B;
    EVENT = DgrAlmEvent;
  };
};
  /* 'external' events for TASK_A */
EVENT ExtEvent1 { MASK = 0x02 };
  /* 'internal' events for TASK_A */
EVENT XEvent { MASK = 0x80 };
EVENT YEvent { MASK = 0x40 };
EVENT Z1_FLG { MASK = 0x20 };
EVENT Z2_FLG { MASK = 0x10 };
  /* event for TASK_B */
EVENT DgrAlmEvent { MASK = 0x01 };
  /* this EVENT object defines two different */
  /* events for TASK_A and TASK_B (see references */
  /* in the TASK objects), but these events */
  /* have one and the same mask */
EVENT ExtEvent2 { MASK = 0x04 };

MESSAGE Norm {
  TYPE = UNQUEUED;
  CDATATYPE = "int";
};
...
```

The C-language file:

```
TASK( TASK_A ) /* Extended task TASK_A */
{
        /* 'external' events ExtEvent1 and ExtEvent2 */
        /* aa is an 'OR' of the masks of these events */
EventMaskType aa = (ExtEvent1 | ExtEvent2);
```

```
        /* 'internal' events (flags) */
EventMaskType x, z1 = Z1_FLG, z2 = Z2_FLG;
int speed;
...
        /* Check the variable and set internal flag if needed */
if (speed == LIMIT)
  {
    x = XEvent;
    SetEvent( TASK_A, x );
  }
...
GetEventMask( TASK_A, &x );  /* check 'internal' flag */
        /* Perform some actions in accordance with */
        /* internal flag status */
if ( (x & XEvent) != 0 ) ClearEvent( z1 );
else SetEvent( TASK_A, z2 );
if ( (x & YEvent) == 0 ) ChainTask( TASK_C );
...
        /* the task is stopped until one of three */
        /* 'external' events is set by another task */
WaitEvent( aa );
        /* clear all 'external' events after awakening  */
ClearEvent( aa );
   ...
}


TASK( TASK_B )               /* Extended task TASK_B */
{
EventMaskType b_ev, a_ev;
b_ev = DgrAlmEvent | ExtEvent2;
InitCounter( DgrCnt, 0 ); /* initialize the counter */
...
                        /* this alarm will awake this task */
SetRelAlarm( AWAKE, 20, 0 );
WaitEvent( b_ev );       /* waiting for one of two events */

/* The task will be ready again when one of two */
/* events are set. One of them - DgrAlmEvent will */
/* be set by the alarm AWAKE after 20 ticks of the */
/* counter DgrCnt. Thus, the task can delay itself. */
```

```
ClearEvent( b_ev );          /* clear all events */
GetEvent( TASK_A, &a_ev );  /* get events of TASK_A */
if ( (a_ev & ExtEvent1) == 0)
  {
    a_ev = ExtEvent2;
    SetEvent( TASK_A, a_ev );
  }                 /* set the event ExtEvent2 for TASK_A */
...
}

TASK( TASK_C )       /* Basic task TASK_C */
{
EventMaskType bb, set;
set = ExtEvent2;
...
GetEvent( TASK_B, &bb );
                    /* if the event ExtEvent2 for TASK_B */
                    /* is clear, set it */
if ( (bb & ExtEvent2) == 0 ) SetEvent( TASK_B, set );
...
}
```

Counter Management Services

Data Types and Identifiers

The following data types are established by OSEK OS to work with counters:

- *TickType*– the data type represent count value in ticks
- *TickRefType*– the data type of a pointer to the variable of data type *TickType*
- *CtrRefType*– the data type references a counter
- *CtrInfoRefType* – the data type of a pointer to the structure of data type *CtrInfoType*
- *CtrInfoType*– the data type represents a structure for storage of counter characteristics. This structure has the following elements:

    – *maxallowedvalue* – maximum possible allowed counter value in ticks

    – *ticksperbase* – number of ticks required to reach a counter-specific significant unit

    – *mincycle* – minimum allowed number of ticks for the cycle parameter of *SetRelAlarm* and *SetAbsAlarm* services (only for system with Extended Status)

All elements of *CtrInfoType* structure have the data type *TickType*, and the structure looks like the following:

```
/* for EXTENDED status */
typedef CtrInfoType tagCIT;
struct tagCIT
{
    TickType maxallowedvalue;
    TickType ticksperbase;
    TickType mincycle;
};

/* for STANDARD status */
typedef CtrInfoType tagCIT;
struct tagCIT
{
    TickType maxallowedvalue;
    TickType ticksperbase;
};
```

NOTE   CtrRefType, CtrInfoType and CtrInfoRefType data types are not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. These are OSEKturbo extension of OSEK OS.

Constants

For all counters, the following constants are defined:

- *OSMAXALLOWEDVALUE_cname*
  Maximum possible allowed value of counter <cname> in ticks.

- *OSTICKSPERBASE_cname*
  Number of ticks required to reach a specific unit of counter <cname>.

- *OSMINCYCLE_cname*
Minimum allowed number of ticks for a cyclic alarm of counter
<cname>. This constant is not defined in *STANDARD* status

For system counters, which are always time counters, the special
constants are provided by the operating system:

- *OSMAXALLOWEDVALUE / OSMAXALLOWEDVALUE2*
maximum possible allowed value of the system/second timer in
ticks (see also "Counter Definition")

- *OSTICKSPERBASE / OSTICKSPERBASE2*
number of ticks required to reach a counter-specific value in the
system/second counter (see also "Counter Definition")

- *OSTICKDURATION / OSTICKDURATION2*
duration of a tick of the system/second counter in nanoseconds
(defined automatically by *System Generator utility* (see also "CPU
Related Attributes")

- *OSMINCYCLE / OSMINCYCLE2*
minimum allowed number of ticks for a cyclic alarm attached to
the system/second counter (only for system with Extended
Status, see also "Alarm Definition")

NOTE   *OSMAXALLOWEDVALUE2*, OSTICKSPERBASE2,
OSTICKDURATION2, and OSMINCYCLE2 constants are not
defined in the *OSEK/VDX Operating System, v.2.2, 10 September
2001* specification. These are OSEKturbo extension of OSEK OS.

Counter Declaration

The declaration statement *DeclareCounter* may be used for
compatibility with previous OSEK versions. It may be omitted in
application code.

Syntax: `DeclareCounter( <name of counter> );`

Input: *<name of counter>* – a reference to the counter.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are
defined at system generation phase.

Conformance: BCC1, ECC1

InitCounter

Syntax:

```
StatusType InitCounter( CtrRefType <CounterID>,
                        TickType <Ticks> );
```

Input: *<CounterID>* – a reference to the counter.

<Ticks> – a counter initialization value in ticks.

Output: None.

Description: Sets the initial value of the counter with the value *<Ticks>*. After this call the counter will advance this initial value by one via the following call of *CounterTrigger*. If there are running attached alarms, then their state stays unchanged, but the expiration time become indeterminate.

Particularities: The service call is allowed on task level only.

This service is not implemented if no counters are defined in the configuration file.

The *InitCounter* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status: • Standard:

– E_OK – no error.

• Extended:

– E_OS_ID – the counter identifier is invalid.

– E_OS_VALUE – the counter initialization value exceeds the maximum admissible value.

– E_OS_CALLEVEL – a call at interrupt level (not allowed).

Conformance: BCC1, ECC1

CounterTrigger

Syntax: `StatusType CounterTrigger( CtrRefType <CounterID> );`

Input: *<CounterID>* – a reference to the counter.

Output: None.

Description: The service increments the current value of the counter. If the counter value was equal to *maxallowedvalue* (see "Data Types and Identifiers") it is reset to zero.

If alarms are linked to the counter, the system checks whether they expired after this tick and performs appropriate actions (task activation and/or event setting).

Particularities: The service call is allowed on task level and ISR level, but not in hook routines.

This service may be used only on software counters. It has no sense to use this service on a hardware counter.

This service is not implemented if no counters are defined in the configuration file.

The *CounterTrigger* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the counter identifier is invalid or belongs to hardware counter.

Conformance: BCC1, ECC1

GetCounterValue

Syntax:
```
StatusType GetCounterValue( CtrRefType
<CounterID>, TickRefType <TicksRef> );
```

Input: *<CounterID>* – a reference to the counter.

Output: *<TicksRef>* – a pointer to counter value in ticks. Reference to *TickType* variable can be used instead of *TickRefType* variable.

Description: The system service provides the current value of the counter *<CounterID>* in ticks and saves it in variable referenced by *<TicksRef>*.

Particularities: The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

This service is not implemented if no counters are defined in the configuration file.

The *GetCounterValue* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the counter identifier is invalid.

Conformance: BCC1, ECC1

GetCounterInfo

Syntax: ```
StatusType GetCounterInfo(CtrRefType <CounterID>,
CtrInfoRefType <InfoRef>);
```

Input: *<CounterID>* – a reference to the counter.

Output: *<InfoRef>* – a pointer to the structure of *CtrInfoType* data type with constants of the counter. Reference to the *CtrInfoType* variable can be used instead of the *CtrInfoRefType* variable.

Description: The service provides the counter characteristics into the structure referenced by *<InfoRef>*. For a system counter special constants may be used instead of this service.

Particularities: The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines. The structure referenced by *<InfoRef>* consists of two elements in case of the "Standard Status", and of three elements in case of the "Extended Status".

This service is not implemented if no counters are defined in the configuration file.

The *GetCounterInfo* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the counter identifier is invalid.

Conformance: BCC1, ECC1

## Examples for Counter Management

The example shows how *CounterTrigger*, *GetCounterValue* services can be used.

The following definitions are made in the definition file:

```
...
COUNTER FirstCnt {
  MAXALLOWEDVALUE = 127;
  TICKSPERBASE = 1;
  MINCYCLE = 1;
};
COUNTER SecondCnt {
  MAXALLOWEDVALUE = 36;
  TICKSPERBASE = 1;
  MINCYCLE = 1;
};
ISR First_Handler {
  CATEGORY = 2;
  PRIORITY = 1;
};
TASK TaskCnt {
  PRIORITY = 1;
  SCHEDULER = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
};
...
```

Following C-language code shows how CounterTrigger can be used to increment counter:

```
ISR First_Handler {
...    /* User's code */
  CounterTrigger( FirstCnt );
...    /* User's code */
}


TASK TaskCnt {
TickType cnt1;
...    /* User's code */

CounterTrigger( FirstCnt );
GetCounterValue( FirstCnt, &cnt1 );
if( cnt1 == 0 )
  {
  CounterTrigger( SecondCnt );
  }
...    /* User's code */
}
```

Alarm Management Services

Data Types and Identifiers

The following data types are established by OSEK OS to work with alarms:

- *TickType*– data type represents count values in ticks
- *TickRefType*– the data type of a pointer to the variable of data type *TickType*
- *AlarmBaseType*– the data type represents a structure for storage of counter characteristics. The elements of the structure are:
  - *maxallowedvalue* – maximum possible allowed counter value in ticks
  - *ticksperbase* – number of ticks required to reach a counter-specific significant unit

– *mincycle* – minimum allowed number of ticks for the cycle parameter of *SetRelAlarm* and *SetAbsAlarm* services (only for system with Extended Status)

All elements of the structure are of data type *TickType*.

- *AlarmBaseRefType*– the data type references data corresponding to the data type *AlarmBaseType*
- *AlarmType*– the data type represents an alarm object

Constants

OSMINCYCLE – minimum allowed number of ticks for a cyclic alarm (only for system with Extended Status)

Alarm Declaration

The declaration statement *DeclareAlarm* may be used for compatibility with previous OSEK versions. It may be omitted in application code.

Syntax: `DeclareAlarm( <name of alarm> );`

Input: *<name of alarm>* – a reference to the alarm.

Description: This is a dummy declaration.

Particularities: There are no need for this declaration because all system objects are defined at system generation phase.

Conformance: BCC1, ECC1

GetAlarmBase

Syntax: `StatusType GetAlarmBase( AlarmType <AlarmID>,`
`                    AlarmBaseRefType <InfoRef> );`

Input: *<AlarmID>* – a reference to the alarm.

Output: *<InfoRef>* – a pointer to the structure *<InfoRef>* with returned values of the alarm base. Reference to *AlarmBaseType* variable can be used instead of *AlarmBaseRefType* variable.

Description: The service returns the alarm base characteristics into the structure pointed by *<InfoRef>*. The return value is a structure in which the information of data type *AlarmBaseType* is stored.

Particularities: The structure consists of two elements in case of the "Standard Status", and of three elements in case of the "Extended Status".

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

This service is not implemented if no alarms are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.
- Extended:
  - E_OS_ID – the alarm identifier is invalid.

Conformance: BCC1, ECC1

GetAlarm

Syntax:
```
StatusType GetAlarm( AlarmType <AlarmID>,
                     TickRefType <TicksRef> );
```

Input: *<AlarmID>* – a reference to the alarm.

Output: *<TicksRef>* – a pointer to a variable which gets a relative value in ticks before the alarm expires. Reference to *TickType* variable can be used instead of *TickRefType* variable.

Description: This service calculates the time in ticks before the alarm expires. If the alarm is not in use, then returned value is not defined.

Particularities: It is up to the application to decide whether for example an alarm may still be useful or not.

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

This service is not implemented if no alarms are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.
  - E_OS_NOFUNC – the alarm is not in use.
- Extended:

– E_OS_ID – the alarm identifier is invalid.

Conformance: BCC1, ECC1

SetRelAlarm

Syntax: `StatusType SetRelAlarm( AlarmType <AlarmID>,`
`        TickType <Increment>, TickType <Cycle> );`

Input: *<AlarmID>* – a reference to the alarm;

<Increment> – an alarm initialization value in ticks;

<Cycle> – an alarm cycle value in ticks in case of cyclic alarm. In case of single alarms, the value cycle has to be equal zero.

Output: None.

Description: The system service occupies the alarm *<AlarmID>* element. After *<Increment>* counter ticks have elapsed, the task assigned to the alarm *<AlarmID>* is activated or the assigned event (only for Extended Tasks) is set.

If *<Cycle>* is unequal to 0, the alarm element is logged on again immediately after expiry with the relative value *<Cycle>*. Otherwise, the alarm triggers only once.

If relative value *<Increment>* equals 0, the alarm expires immediately and assigned task becomes *ready* before the system service returns to the calling task or ISR.

Particularities: Allowed on task level and ISR level, but not in hook routines.

If alarm is already in use, the service call is ignored. To change values of alarms already in use the alarm has to be cancelled first.

This service is not implemented if no alarms are defined in the configuration file.

Status: • Standard:
  – E_OK – no error.
  – E_OS_STATE – the alarm is already in use.
  • Extended:
  – E_OS_ID – the alarm identifier is invalid.

– E_OS_VALUE - an alarm initialization value is outside of the admissible limits (lower than zero or greater than the maximum allowed value of the counter), or alarm cycle value is unequal to 0 and outside of the admissible counter limits (less than the minimum cycle value of the counter or greater than the maximum allowed value of the counter).

Conformance: • BCC1, ECC1.

SetAbsAlarm

Syntax:
```
StatusType SetAbsAlarm( AlarmType <AlarmID>,
             TickType <Start>, TickType <Cycle> );
```

Input: *<AlarmID>* – a reference to the alarm;

<Start> – an absolute value in ticks;

<Cycle> – an alarm cycle value in ticks in case of cyclic alarm. In case of single alarms, cycle has to be equal zero.

Output: None.

Description: The system service occupies the alarm *<AlarmID>* element. When *<Start>* ticks are reached, the task assigned to the alarm *<AlarmID>* is activated or the assigned event (only for Extended Tasks) is set.

If *<Cycle>* is unequal to 0, the alarm element is logged on again immediately after expiry with the relative value *<Cycle>*. Otherwise, the alarm triggers only once.

If the absolute value <Start> is very close to the current counter value, the alarm may expire and assigned task may become *ready* before the system service returns to the calling task or ISR.

If the absolute value <Start> have been reached before the service call, the alarm will only expire when <Start> value will be reached again.

Particularities: Allowed on task level and ISR level, but not in hook routines.

If alarm is already in use, the service call is ignored. To change values of alarms already in use the alarm has to be cancelled first.

This service is not implemented if no alarms are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error;
  - E_OS_STATE – the alarm is already in use.
- Extended:
  - E_OS_ID – the alarm identifier is invalid.
  - E_OS_VALUE - an alarm absolute value is outside of the admissible limits (lower than zero or greater than the maximum allowed value of the counter), or alarm cycle value is unequal to 0 and outside of the admissible counter limits (less than the minimum cycle value of the counter or greater than the maximum allowed value of the counter).

Conformance:
- BCC1, ECC1.

CancelAlarm

Syntax: `StatusType CancelAlarm( AlarmType <AlarmID> );`

Input: *<AlarmID>* – a reference to the alarm.

Output: None.

Description: The service cancels the alarm (transfers it into the stop state).

Particularities: The service is allowed on task level and in ISR, but not in hook routines.

This service is not implemented if no alarms are defined in the configuration file.

Status:
- Standard:
  - E_OK – no error.
  - E_OS_NOFUNC – the alarm is not in use.
- Extended:
  - E_OS_ID – the alarm identifier is invalid.

Conformance: BCC1, ECC1

StartTimeScale

Syntax:
```
void StartTimeScale ( void );
```

Input: None.

Output: None.

Description: The service starts a *TimeScale* processing. The first Task in *TimeScale* is activated immediately, the subsequent tasks are activated in according with *StepTime* and *StepNumber* values.

Particularities: Allowed on task level only.

The *StartTimeScale* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status:
- Standard:
  - None.
- Extended:
  - None.

Conformance: BCC1, ECC1

StopTimeScale

Syntax:
```
void StopTimeScale ( void );
```

Input: None.

Output: None.

Description: The service cancels a TimeScale processing by disabling interrupts from the system timer.

Particularities: Allowed on task level, on ISR level and in all hook routines.

The *StopTimeScale* service is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Status:
- Standard:
  - None.
- Extended:

– None.

Conformance: BCC1, ECC1

Examples for Alarm Management

The example shows how counters and alarms can be used. For examples of *TimeScale* using see OSEKturbo User's Guide, 3.8 *TimeScale*. Also the sample application can be used as an example for *TimeScale*.

The following definitions are made in the definition file:

```
...
TASK TaskTime {
  PRIORITY = 3;
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
};
TASK TASK_B {
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  STACKSIZE = 256;
  EVENT = DgrAlmEvent;
};
TASK TASK_X {
  PRIORITY = 1;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
};
COUNTER TimeCnt {
  MAXALLOWEDVALUE = 127;
  TICKSPERBASE = 1;
  MINCYCLE = 1;
};
COUNTER DgrCnt {
  MAXALLOWEDVALUE = 36;
  TICKSPERBASE = 1;
  MINCYCLE = 1;
```

```
};
ALARM TimeAlm {
  COUNTER = TimeCnt;
  ACTION = ACTIVATETASK {
    TASK = TASK_X;
  };
};
ALARM DgrAlm {
  COUNTER = DgrCnt;
  ACTION = SETEVENT {
    TASK = TASK_B;
    EVENT = DgrAlmEvent;
  };
};
EVENT DgrAlmEvent {
  MASK = 0x01;
};

MESSAGE Norm {
  TYPE = UNQUEUED;
  CDATATYPE = "int";
};
...
```

The alarm TimeAlm activates the task `TASK_X` when the counter `TimeCnt` expires. The alarm `DgrAlm` sets the specified event for the task `TASK_B` when the counter `DgrCnt` expires.

The C-language code can be the following:

```
OSMSGNorm _Norm;

TASK( TaskTime )
{
TickType curTime;
TickType ticksToExpire;
OSBYTE i=0;

InitCounter( TimeCnt, 0 ); /* init time counter with 0 value */
CounterTrigger( TimeCnt ); /* increments counter */

while (i != 1) {
```

```
        /* read TimeCnt value */
  GetCounterValue( TimeCnt, &curTime );
  if( curTime == CONST )
  {     /* if desired value, activate TaskB */
    ActivateTask( TASK_B );
    SetRelAlarm( TimeAlm, 20, 0 );
        /* activate TaskX when TimeCnt reaches */
    GetAlarm( TimeAlm, &ticksToExpire );
        /* just for example: TimeAlm will */
    /* expire after 'ticksToExpire' ticks of TimeCnt */
  }
    /* if more than desired value, terminate the task */
  if( curTime > CONST ) TerminateTask();
}
}


TASK( Task_B )
{
OSMSGNorm _Norm;
EventMaskType evMask;

evMask = DgrAlmEvent;
                    /* init degree counter with 0 value */
InitCounter( DgrCnt, 0 );
SetAbsAlarm( DgrAlm, 75, 15 ); /* set cyclic alarm */
WaitEvent( evMask );
            /* wait for event which must be set by the alarm */
_Norm = 1;  /* wake up and send the message that all is OK */
SendMesssage( Norm, _Norm);
TerminateTask();
}


ISR( Timer_Isr )
{
...  /* reset the hardware */
      /* increment the counter and process */
      /* alarms attached to the counter */
CounterTrigger( TimeCnt );
}


ISR( Dgr_Isr )
{
```

For More Information: www.freescale.com

```
...    /* reset the hardware */
       /* increment the counter and process */
       /* alarms attached to the counter */
CounterTrigger( DgrCnt );
}
```

Communication Management Services

Data Types and Identifiers

The following names are used in the OSEK Operating System for work with messages:

- *SymbolicName* – an unique name representing a message. It only can be used in conjunction with calls of the message service. A *SymbolicName* need not be a data type. Variables or constants of *SymbolicName* can be declared or used

- *AccessName* – an unique name defining access to a message object. Depending on the chosen configuration, a distinction is made between the following *AccessName* scheme:

  WithCopy configuration:

  A application variable exists as a local copy of the message. The name of the variable is the *AccessName*. This variable contains a copy of the corresponding message object

  WithoutCopy configuration:

  The message object data is accessed via the *AccessName*. This *AccessName* is a static link: it refers directly to the message object data. The *AccessName* refers to the same data (RAM) as the message object

- *AccessNameRef* – the address of the message data field

- *FlagType* – the abstract data type for flag identification

SendMessage

Syntax:
```
StatusType SendMessage ( SymbolicName <Message>,
                         AccessNameRef <Data> );
```

Input: *<Message>* – symbolic name of the message object.

<Data> – the reference to the message data to be sent.

Output: None.

Description: *Unqueued and Queued WithCopy message*:

This service updates the message object *<Message>* with the Data given by *<Data>* and requests the transmission of the message object to the receiver. The message object is locked during this update and during the transmission of the data to the underlying communication layer. The message object is not updated if it is already locked. This service updates also the status information of the message object accordingly.

Unqueued WithoutCopy message:

This service requests the transmission of the already updated message object to the receiver.

Particularities: Allowed for all message types on task level and in message's callback function. Unqueued messages in case of *WithCopy* only are allowed on ISR level and in *ErrorHook* routine.

SendMessage does not verify whether the message object has been initialized prior to sending it.

This service is not implemented if no messages are defined in the configuration file.

Status: • Standard:

– E_OK – no error.

– E_COM_LOCKED – message object locked (other task or ISR is sending or receiving the same message or this message resource is occupied).

– E_OS_CALLEVEL - call at ISR level for queued message or with *<Data>* configured WithOutCopy; this status is OSEKturbo extension of OSEK OS.

• Extended:

– E_COM_ID – invalid parameter *<Message>* or sending message other than unqueued one in *WithCopy* configuration from ISR level.

Conformance: BCC1, ECC1

ReceiveMessage

Syntax:
```
StatusType ReceiveMessage ( SymbolicName
<Message>,AccessNameRef <Data>);
```

Input: *<Message>* – symbolic name of the message object.

Output: *<Data>* – the reference to the message data to be received.

Description: *Unqued and Queued WithCopy message*:

This service delivers the message data associated with the message object *<Message>* to the applications message copy referenced by *<Data>*. The message object is locked during data reception. This service also updates the status information of the message object accordingly.

Unqueued WithoutCopy message:

Returns Status only since application can access directly to the message object.

Particularities: Unqueued messages:

- The service returns the current message.
- If no new message has been received since the last call to *ReceiveMessage* the current message is returned.

Allowed for all message types on task level and in message's callback function. Unqueued messages in case of *WithCopy* only are allowed on ISR level and in *ErrorHook* routine.

Status:
- Standard:
    - E_OK – no error.
    - E_COM_LOCKED – message object locked (other task or ISR is sending or receiving the same message or this message resource is occupied).
    - E_COM_NOMSG - the queued message identified by *<Message>* is empty.
    - E_COM_LIMIT - an overflow of the FIFO of the queued message identified by *<Message>* occurred since the last call to ReceiveMessage for that particular *<Message>*.

– E_OS_CALLEVEL - call at ISR level for queued message
or with <Data> configured WithOutCopy; this status is
OSEKturbo extension of OSEK OS.

- Extended:

– E_COM_ID – invalid parameter *<Message>* or receiving
message other than unqueued one in *WithCopy*
configuration from ISR level).

Conformance: BCC1, ECC1

GetMessageResource

Syntax:
```
StatusType GetMessageResource (
                SymbolicName <Message> )
```

Input: *<Message>* – symbolic name of the message object.

Output: None.

Description: The service *GetMessageResource* set the message object *<Message>* status as busy.

Particularities: It is recommended that corresponding calls to *Get*- and *ReleaseMessageResource* should appear within the same function on the same function level. Before terminating the task or entering the wait state the corresponding service *ReleaseMessageResource* shall be called by the application layer.

This service can only be used to support the transfer of a message identified by *<Message>* whose copy configuration is *WithoutCopy*.

Status:
- Standard:

– E_OK - the message has been set to BUSY successfully.

– E_COM_LOCKED - the message is locked.

– E_COM_BUSY - the message is already set to BUSY.

– E_OS_NOFUNC - the message is queued; this status is OSEKturbo extension of OSEK OS.

- Extended:

– E_COM_ID - the message parameter is invalid.

Conformance: BCC1, ECC1

ReleaseMessageResource

Syntax:
```
StatusType ReleaseMessageResource (
                      SymbolicName <Message> )
```

Input: *<Message>* – symbolic name of the message object.

Output: None.

Description: The service *ReleaseMessageResource* shall unconditionally set the message object *<Message>* to NOT_BUSY.

Particularities: It is recommended that corresponding calls to *Get*- and *ReleaseMessageResource* appear within the same function on the same function level. Before terminating the task or entering the wait state the corresponding service *ReleaseMessageResource* shall be used.

This service can only be used to support the transfer of a message identified by *<Message>* whose copy configuration is *WithoutCopy*.

Status:
- Standard:
  - E_OK - the message has been set to NOT_BUSY status.
- Extended:
  - E_COM_ID - the message parameter is invalid.
  - E_OS_NOFUNC - the message is queued; this status is OSEKturbo extension of OSEK OS.

Conformance: BCC1, ECC1

GetMessageStatus

Syntax:
```
StatusType GetMessageStatus (
                      SymbolicName <Message> )
```

Input: *<Message>* – symbolic name of the message object.

Output: None.

Description: The service *GetMessageStatus* return the current status of the message object *<Message>*. If this service call fails, it return an implementation specific error code that shall be distinguishable from all other return values.

Particularities: None.

Status:
- Standard:
  - E_COM_LOCKED - the message is locked.
  - E_COM_BUSY - the message is currently set to BUSY.
  - E_COM_NOMSG - the FIFO of the queued message identified by *<Message>* is empty.
  - E_COM_LIMIT - an overflow of the FIFO of the queued message identified by *<Message>* occurred.
  - E_OK - none of the above conditions is applicable and no indication of error is present.
- Extended:
  - E_COM_ID - the *<Message>* parameter is invalid.

Conformance: BCC1, ECC1

StartCOM

Syntax: `StatusType StartCOM (void);`

Input: None.

Output: None.

Description: This service calls message initialization routines.

Particularities: This routine performs the initialization of the application specific message objects by calling the *MessageInit* function provided by the application programmer. *StartCOM* is called from within a task.

Status:
- E_OK – the initialization completed successfully.
- The service returns an implementation or application specific error code if the initialization did not complete successfully.

Conformance: BCC1, ECC1

InitCOM

Syntax: `StatusType InitCOM (void);`

Input: None.

Output: None.

Description: The service is intended for initialization of communication hardware.

Particularities: This service does nothing in OSEKturbo.

Status: – E_OK – the initialization completed successfully.

Conformance: BCC1, ECC1

CloseCOM

Syntax: `StatusType CloseCOM (void);`

Input: None.

Output: None.

Description: The service is intended for de-initialization of communication hardware.

Particularities: This service does nothing in OSEKturbo.

Status: – E_OK – the initialization completed successfully.

Conformance: BCC1, ECC1

StopCOM

Syntax: `StatusType StopCOM (Scalar <ShutdownMode>);`

Input: Input parameter is indifferent in case of local communication.

Output: None.

Description: This service is used to terminate a session of OSEK COM, release resources where applicable.

Status: – E_OK – no error.

– E_COM_BUSY - application uses COM resources.

Conformance: BCC1, ECC1

MessageInit

Syntax: `StatusType MessageInit (void);`

Input: None.

Output: None.

Description: This routine initializes all application specific message objects.

Particularities: This function is provided by the application programmer and is called by the *StartCOM* routine only.

Status:
- E_OK – if the initializationinitialization of the application specific message object has completed successfully.
- The service returns an implementation or application specific error code if the initialization did not complete successfully.

Conformance: BCC1, ECC1

ReadFlag

Syntax: `FlagValue ReadFlag (FlagType <FlagName>)`

Input: *<FlagName>* – message flag name.

Output: FlagValue – state of the flag *<FlagName>*.

Description: This service returns the value of the specified notification flag.

Particularities: This service returns 0 if the flag is not set, i.e. message has not arrived.

Status:
- Standard and Extended:
  - TRUE - the flag is set by arrived message.
  - FALSE - the flag is not set.

Conformance: BCC1, ECC1

ResetFlag

Syntax: `StatusType ResetFlag (FlagType <FlagName>)`

Input: *<FlagName>* – message flag name.

Output: None.

Description: This service set the specified notification flag to FALSE.

Particularities: This service is provided to ensure flag data consistency thus enabling portable access to the message notification flag.

Status: • Standard and Extended:

– E_OK - the flag reset completed successfully.

Conformance: BCC1, ECC1

Examples of Using Messages

The examples below present the usage of system services for communication. The following definitions can be made in the definition file:

```
OS os1 {
  ...
  MessageCopyAllocation = USER;
  ...
};
TASK TASK_A {
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  PRIORITY = 2;
  STACKSIZE = 256;
  EVENT = MYEVENTMSGA;
  ACCESSOR = RECEIVED {
    MESSAGE= MsgA;
    WITHOUTCOPY= FALSE;
    ACCESSNAME= "Msga";
  };
};
TASK TASK_B {
  SCHEDULE = FULL;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  PRIORITY = 3;
  STACKSIZE = 256;
  ACCESSOR = RECEIVED {
    MESSAGE= MsgB;
    WITHOUTCOPY= FALSE;
    ACCESSNAME= "MessageBuffer";
  };
```

```
};
TASK TASK_C {
  SCHEDULE = FULL;
  AUTOSTART = TRUE;
  ACTIVATION = 1;
  PRIORITY = 1;
  STACKSIZE = 256;
  ACCESSOR = SENT {
        MESSAGE= MsgA;
        WITHOUTCOPY= TRUE;
        ACCESSNAME= "msgAbuf";
        };
        ACCESSOR = SENT {
        MESSAGE= MsgB;
        WITHOUTCOPY= FALSE;
        ACCESSNAME= "msgB";
  };
};
MESSAGE MsgA {
  TYPE = UNQUEUED;
  CDATATYPE = "MSGA";
  ACTION = SETEVENT{
      EVENT = MYEVENTMSGA;
      TASK  = TASKA;
  };
};
MESSAGE MsgB {
  TYPE = QUEUED {
      QUEUEDEPTH = 5;
  };
  CDATATYPE = "MSGB";
  ACTION = ACTIVATETASK{
      TASK = TASK_B;
  };
};
EVENT MYEVENTMSGA {
  MASK = AUTO;
};
...
```

The C-language code can be the following:

For More Information: www.freescale.com

```
typedef struct{
    int    msg;
  } MSGA;              /* data type for MsgA */

typedef struct{
    int    num;
    char        data;
  } MSGB;              /* data type for MsgA */

TASK( TASK_A )
{
  MSGA msgA;
  ...
  ClearEvent( MYEVENTMSGA );
  WaitEvent( MYEVENTMSGA );
  ReceiveMessage( MsgA, &msgA ); /* with copy */
      /* received data in msgA may be freely used */
        ...
}

TASK( TASK_B )
{
  int rnum;
  char rdata;
  MSGB msgB;
  ...
  ReceiveMessage( MsgB, &msgB );/* receive with copy */
  rnum = msgB.num;                 /* get data from message */
    rdata = msgB.data;
  ...
  TerminateTask();
}

int msgNum = 0;                          /* number for MsgB */

TASK( TASK_C )
{
  MSGB msgB;
  ...
  msgAbuf.msg = MyIntData;  /* put some data into message */
  msgB.num = msgNum;        /* put number into message copy */
```

```
    msgNum++;
    msgB.data= MyCharData;            /* put data into message copy */
    SendMessage( MsgA, &msgAbuf );/* w/o copy*/
    SendMessage( MsgB, &msgB );   /* with copy */
    ...
}
```

Debugging Services

These services are not defined by *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

GetRunningStackUsage

Syntax: `unsigned short GetRunningStackUsage ( void );`

Input: None.

Output: • amount of stack used by running task in bytes.

          • 0xFFFF if there is not any running task or the task uses "single stack".

Description: The service returns amount of stack used by running task in bytes. The service returns 0xFFFF for basic task because single stack is used.

Particularities: The service is implemented if the value of the *StackOverflowCheck* attribute is *TRUE* or the value of the *DEBUG_LEVEL* attribute is greater than 0.

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

Conformance: BCC1, ECC1

GetStackUsage

Syntax: `unsigned short GetStackUsage ( TaskType <TaskID> );`

Input: *<TaskID>* - a reference to the task.

Output: • amount of stack used by task <TaskID> in bytes.

• 0xFFFF if the task is basic (uses "single stack").

Description: The service returns stack usage by task *<TaskID>* in bytes.

Particularities: The service is implemented if the value of the *StackOverflowCheck* attribute is set *TRUE* or the value of the *DEBUG_LEVEL* attribute is greater than 0.

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

Conformance: BCC1, ECC1

GetTimeStamp

Syntax: `unsigned short GetTimeStamp (void );`

Input: None.

Output: System Counter current value.

Description: The service returns current value of the System Counter (the counter which is attached to the System Timer).

Particularities: The service is implemented if the value of the *DEBUG_LEVEL* attribute is greater than 0 and the System Timer is defined in the application.

The service call is allowed on task level, ISR level and in *ErrorHook*, *PreTaskHook* and *PostTaskHook* hook routines.

Conformance: BCC1, ECC1

Operating System Execution Control

Data Types

The OSEK OS establishes the following data type for operation mode representation:

• StatusType – the data type represent variable for saving system status

• *AppModeType* – the data type represents the operating mode

Constants

The following constant is used within the OSEK OS to indicate default application mode:

- *OSDEFAULTAPPMODE* – constant of data type *AppModeType*. The constant is assigned to one of the application modes defined in the OIL file. This constant is always a valid parameter for *StartOS* service.

The following constants are used within the OSEK Operating System to indicate system status. All of them have type *StatusType*. Status meaning is specified in service descriptions:

- E_OK
- E_OS_ACCESS
- E_OS_CALLEVEL
- E_OS_ID
- E_OS_LIMIT
- E_OS_NOFUNC
- E_OS_RESOURCE
- E_OS_STATE
- E_OS_VALUE
- E_OS_SYS_STACK[1]
- E_OS_SYS_ORDER[1]
- E_OS_SYS_MAINSTACK[1]
- E_OS_SYS_ISRSTACK[1]
- E_COM_ID
- E_COM_LOCKED
- E_COM_BUSY
- E_COM_NOMSG
- E_COM_LIMIT

---

[1] E_OS_SYS_xxx are not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

GetActiveApplicationMode

Syntax: `AppModeType GetActiveApplicationMode( void );`

Input: None.

Output: Current application mode.

Description: This service returns the current application mode.

Particularities: Allowed on task level, on ISR level and in all hook routines.

Conformance: BCC1, ECC1

StartOS

Syntax: `void StartOS( AppModeType <Mode> );`

Input: *<Mode>* – an operating mode.

Output: None.

Description: This service starts the operation system in a specified mode. If a *StartupHook* is configured, the hook routine is always called before starting the application.

Particularities: Allowed outside of the operating system only.
In OSEKturbo this service does not returns to the caller.

Conformance: BCC1, ECC1

ShutdownOS

Syntax: `void ShutdownOS( StatusType <Error> );`

Input: *<Error>* – a code of the error occurred.

Output: None.

Description: The service aborts the overall operating system.

If a task is in the *running* state, *PostTaskHook* is not called.

If a *ShutdownHook* is configured, the hook routine is always called (with *<Error>* as argument) before shutting down the operating system. If *ShutdownHook* returns, the operating system enters

For More Information: www.freescale.com

endless loop with the lowest "Power-down" mode. (see "System Shutdown").

Particularities: *ShutdownOS* never returns.

ShutdownOS runs in connection with the currently active context, which may be unknown to the user. Thus, no API functions are admitted within the *ShutdownHook* routine.

Allowed on task level, on ISR level and in *StartupHook* and *ErrorHook* hook routines.

Conformance: BCC1, ECC1

Hook Routines

ErrorHook

Syntax: `void ErrorHook( StatusType <Error> );`

Input: *<Error>* – a code of the error occurred.

Output: None.

Description: To implement *ErrorHook* the routine with name 'ErrorHook' shall be defined in user's code. This routine is called by the operating system at the end of a system service which has a return value not equal to E_OK if not specified other. It is called before returning to the task level. This hook is also called from OS dispatcher when an error is detected during task activation or event setting as a result of an alarm expiry or a message arrival, or if a rescheduling occurs after *Suspend(OS/All)Interrupts* before call to appropriate Resume function.

Particularities: *ErrorHook* can not be nested. Therefore the error hook is not called, if a system service called from *ErrorHook* and does not return E_OK as a status value.

See "Hook Routines" for general description of hook routines.

This hook is not called if the system configuration option *ERRORHOOK* is set to *FALSE*.

Interrupts category 2 (and interrupts category 1 if their priority is not higher than any ISR category 2 priority) are disabled in *ErrorHook*.

There is a set of special macros to get the ID of service where error has occured and it's first argument - see "Macros for ErrorHook".

Conformance: BCC1, ECC1

PreTaskHook

Syntax: `void PreTaskHook( void );`

Input: None.

Output: None.

Description: To implement *PreTaskHook* the routine with name 'PreTaskHook' shall be defined in user's code. This hook routine is called by the operating system before executing a new task, but after the transition of the task to the running state (to allow evaluation of the task ID by *GetTaskID* services). This hook is called from the scheduler when it passes control to the given task. It may be used by the application to trace the sequences and timing of tasks' execution.

Particularities: Interrupts category 2 (and interrupts category 1 if their priority is not higher than any ISR category 2 priority) are disabled in *PreTaskHook*.

See "Hook Routines" for general description of hook routines.

This hook is not called if the system configuration option *PRETASKHOOK* is set to FALSE.

Conformance: BCC1, ECC1

PostTaskHook

Syntax: `void PostTaskHook( void );`

Input: None.

Output: None.

Description: To implement *PostTaskHook* the routine with name 'PostTaskHook' shall be defined in user's code. This hook routine is called by the operating system after executing the current task, but before leaving

the task's running state (to allow evaluation of the task ID by *GetTaskID*).This hook is called from the scheduler when it switches from the current task to another. It may be used by the application to trace the sequences and timing of tasks' execution.

Particularities: Interrupts category 2 (and interrupts category 1 if their priority is not higher than any ISR category 2 priority) are disabled in *PostTaskHook*.

PostTaskHook is not called if running task is exist and OS is aborted by *ShutdownOS* service.

See "Hook Routines" for general description of hook routines.

This hook is not called if the system configuration option *POSTTASKHOOK* is set to *FALSE*.

Status: None.

Conformance: BCC1, ECC1

StartupHook

Syntax: `void StartupHook( void );`

Input: None.

Output: None.

Description: To implement *StartupHook* the routine with name 'StartupHook' shall be defined in user's code. This hook is called by the operating system at the end of the operating system initialization and before the initialization of Interrupt Sources, System and Second timers and before scheduler starts running. At this point in time the application can initialize hardware, etc.

Particularities: Interrupts category 2 (and interrupts category 1 if their priority is not higher than any ISR category 2 priority) are disabled in *StartupHook*.

See "Hook Routines" for general description of hook routines.

This hook is not called if the system configuration option *STARTUPHOOK* is set to *FALSE*.

Status: None.

Conformance: BCC1, ECC1

ShutdownHook

Syntax: `void ShutdownHook( StatusType <Error> );`

Input: *<Error>* – a code of the error occurred.

Output: None.

Description: To implement *ShutdownHook* the routine with name 'ShutdownHook' shall be defined in user's code. This hook is called by the operating system when the *ShutdownOS* service has been called. This routine is called during the operation system shut down. User can avoid return from the hook to calling level. For example reset signal can be generated in the hook.

Particularities: Interrupts category 2 (and interrupts category 1 if their priority is not higher than any ISR category 2 priority) are disabled in *ShutdownHook.*

See "Hook Routines" for general description of hook routines. This hook is not called if the system configuration option *ShutdownHook* is turned off in the configuration file.

Status: None.

Conformance: BCC1, ECC1

IdleLoopHook

Syntax: `void IdleLoopHook( void );`

Input: None.

Output: None.

Description: To implement *IdleLoopHook* the routine with name 'IdleLoopHook' shall be defined in user's code. This hook is called by the operating system from scheduler idle loop. It is not possible to call OSEK OS services from this hook. Hardware dependent code like manipulation with COP registers may be placed here.

Particularities: Interrupts inside this routine are enabled.

IdleLoopHook is executed periodically if *HCLowPower* is set to FALSE. Otherwise IdleLoopHook is executed one time only and CPU transfers itself to the low power mode.

It is recommended to set *HCLowPower* to FALSE if the user enters one of low power modes inside *IdleLoopHook*.

See "Hook Routines" for general description of hook routines.

This hook is not called if the system configuration option *IdleLoopHook* is turned off in the configuration file.

IdleLoopHook hook is not defined in the *OSEK/VDX Operating System, v.2.2, 10 September 2001* specification. This is OSEKturbo extension of OSEK OS.

Conformance: BCC1, ECC1

17

Debugging Application

This chapter provides information about OSEK OS feature intended for debugging a user application with OS.

This chapter consists of the following sections:

- General
- Using OS Extended Status for Debugging
- Context Switch Monitoring
- Stack Debugging Support

General

OSEK OS contains several mechanisms which help user to debug application.

Extended Status

Extended status allows to check most of errors caused by improper use of OSEK services.

ORTI

The purpose of OSEK Run Time Interface (ORTI) implementation is giving the user extended opportunities in debugging embedded OSEK applications. The OSEK Run Time Interface implementation confirms *ORTI: OSEK Run Time Interface, v.2.0 (Draft c), 23 June 1999*; *OSEK/VDX OSEK Run Time Interface (ORTI), Part A: Language Specification, v. 2.1.1, 4 March 2002* and *OSEK/VDX OSEK Run Time Interface (ORTI), Part B: OSEK Objects and Attributes, v. 2.1, 17 April 2002.* The current version of OSEKturbo supports 2.0 and 2.1 ORTI versions.

The ORTI shall be supported from both sides: an OSEK OS and a debugger. The debugger able to display information in terms of

OSEK system objects is "OSEK aware" debugger. The internal OS data is to be made available to the debugger. For this purpose special ORTI file is generated at configuration time by a System Generator. As a result, more information will be available to the user during application debugging session.

System Generator (SysGen) uses OIL file (App.oil) as an input file. Option -**o** of the SysGen defines output ORTI file name. Option -**O** of the SysGen defines the version of ORTI specification to be supported in the output ORTI file. The SysGen utility generates static information in the ORTI format. This utility analyzes the application configuration and generates ORTI file. The same OIL file is used for configuring OS. After application is compiled and linked and executable and map files are created then they are loaded by the debugger. If the debugger is OSEK aware then it can load also the ORTI file for this application. The information from ORTI file provides the debugger with possibility to display information about system objects of current implementation of OSEK OS. This process is depicted on Figure 17.1.

Figure 17.1   Application Building Process with ORTI Support

For More Information: www.freescale.com

Stack Debugging Support

To provide stack usage control the bottom and top stack labels for Extended Tasks are realized in OSEK OS. These labels can be seen in symbol debugger during application execution. They can be used for dynamic control of task stacks usage.

Using OS Extended Status for Debugging

It is strongly recommended to use Operating System Extended Status when you develop an application to analyze return codes of system services. Such OS configuration is more memory and time consuming but it allows the user to save time for errors eliminating. Error codes returned by the OSEK OS services covers most of possible errors that can arise during development. Therefore it is useful to check these codes after a service call to avoid error that can lead to the system crash. For example, a task can perform the *TerminateTask* service while it is still occupying a resource. This service will not be performed and the task will remain active (*running*). In case of Extended Status the *E_OS_RESOURCE* error code is returned and it is possible to detect this situation. But in the system without Extended Status there is no additional check and this error is not indicated and the application behavior will be unpredictable!

When all errors in an application will be eliminated you may turn off the Extended Status and remove additional status checks from the application to get the reliable application of the smaller size.

Context Switch Monitoring

"ORTI Trace Interfaces" provides the user with the ability to trace application execution, including context switching. But there is an ability to monitor context switching without using ORTI. Breakpoints, traces and time stamps can be integrated individually into the application software with the help of context switch hook routines *PreTaskHook* and *PostTaskHook*.

Additionally, the user can set time stamps enabling him to trace the program execution, for example, at the following locations before calling operating system services:

- When activating or terminating tasks;
- When setting or clearing events in the case of Extended Tasks;
- At explicit points of the schedule (*ChainTask*, *Schedule*);
- At the beginning or the end of ISR;
- When occupying and releasing resources or at critical locations.

ORTI Features

ORTI provides two kinds of interfaces to OSEK OS data:

- Trace interface, which means getting an access to the data on a running target when it is essential to trace data changes in a real time;
- Breakpoint interface, which provides an access to desirable data on a stopped target.

Note that ORTI Trace interface is designed to provide access to requested data in accomplished form that is not requiring an additional processing.

ORTI Trace Interfaces

There are trace ORTI interfaces in run-time: running task identification, ISR identification, OSEK OS system calls identification. This interface is turned on, if DEBUG_LEVEL = 2 or greater.

The special attributes are generated for the OS object in ORTI file to support trace interfaces, the names of these attributes corresponds to ORTI specification supported. The following trace interfaces are available to the user:

- running task

  This attribute specifies the name of the currently running task within the OS.

  The value of this attribute presents a one byte memory block which contains either byte numeric identifier of a task which

is currently in the *running* state or the special byte value in case of none tasks are in the *running* state. The certain values of task identifiers are statically determined and enumerated in the type field of running task attribute of an OS object as well as the special value (*NO_TASK*) representing none of tasks running.

The value of the running task attribute is updated each time CPU switches to another task or to the code corresponding to none of task running.

- running ISR

This attribute specifies the identifier of currently executed ISR of category 2. It occupies a one byte memory block.

There are the following special values of the attribute:

- NO_ISR - OS works on task or dispatcher level.
- ISR_LEVEL - OS works on ISR level, but nested interrupt was finished.
- SYSTEM_TIMER - System Timer ISR was activated.
- SECOND_TIMER - Second Timer ISR was activated.

NOTE If there are *SystemTimer* and *SecondTimer* configured both as a *HWCOUNTER* then one System Timer ISR is used for both, as they are sharing the same interrupt request.

Other values correspond to ISRs of category 2 defined in the OIL file.

The next scheme is used for ISR tracing: running ISR attribute has TASK_LEVEL value when OS works at task level. When one of ISRs category 2 starts execution, the value of running ISR is changed to ISR identifier. When SystemTimer ISR gets CPU, the value of running ISR is changed to SYSTEM_TIMER. When ISR terminates and OS returns to task level, running ISR gets value TASK_LEVEL. If nested ISR terminates and OS returns to previous ISR, the running ISR is changed to ISR_LEVEL.

- current service

This attribute specifies which operating service, if any, is currently being executed.

The current service attribute value presents a one byte memory block which represents a service numeric identifier.

This value is updated with OSEK OS service (*Service*) identifier with least signicant bit set when CPU enters *Service* code.

In case of leaving OSEK OS service the attribute value is set to service indentifier with least signicant bit cleared thus indicating end of service. The attribute is updated with such value on the following events occurred:

– CPU leaves OSEK OS service code and starts to execute non OSEK OS service code - i.e. application code in the same or another task;

– CPU leaves OSEK OS service code and starts to execute idle loop.

Typically trace sequence of current service looks like the following:

Table 17.1   ORTI Trace Sequence

Traced Value	Description
...	
SERVICE_1_ID	entering Service 1
SERVICE_1_EXIT	leaving Service 1 (possibly due to task switching)
...	
SERVICE_2_ID	entering Service 2
SERVICE_3_ID	entering nested Service 3 call (e.g. hook routine)
SERVICE_3_EXIT	leaving Service 3 and resuming Service 2
SERVICE_2_EXIT	leaving Service 2 (possibly due to task switching)
...	

ORTI Breakpoint Interface

There is the ORTI Breakpoint interface intended to facilitate debugger access to task related data. The interface is turned on, if DEBUG_LEVEL OIL attribute does not equal to 0.

The following static (at breakpoints) ORTI services are supported for a debugger on breakpoints: access to tasks, stacks, counters, alarms, resources and messages information.

Information needed to display the current status of objects is available for the debugger whenever the debugger is stopped (i.e.

this information is not required in real-time and hence can be read from the TCB or similar structure).

Information in the ORTI file allows a debugger to display task information using values that the user sets in the OIL file.

The following information about OS is available to the user:

- current application mode

  This attribute specifies the current application mode.

  The current application mode value presents a one byte memory block which represents a mode identifier.

- running task priority

  This attribute specifies the current priority of the task reffered by the running task attribute.

  The value of this attribute is provided with taking into account possible task priority changes due to a dynamic resource allocation.

- last error

  This attribute specifies the code of the last error detected. The last error attribute value presents a one byte memory block which represents an error code identifier. This value is updated with error identifier when error occurs. At start up the error code is initialized with E_OK. It is never set back to E_OK after first error

The following task information is available to the user:

- property

  The task property indicates static properties of the task.

- priority

  The task priority value is provided with taking into account possible task priority changes due to a dynamic resource allocation.

- task state

  The task state indicates a standard OSEK state the task is currently in.

- task stack (in ECC1 class)

  The task stack shows which stack area is used by the task.

- event states

  The event states can be used to determine events which are currently set or cleared.

- wait events

  Wait events value contains bit set to one for event which task is waiting for.

- event masks

  The event masks define correspondence between event name and bit mask.

The following stack information is available to the user:

- size

  The size specifies the size (in bytes) of a memory area allocated for stack.

- base address

  The base address specifies the lowest address of stack memory area.

- stack direction

  The stack direction specifies the direction of stack growth and may have "UP" or "DOWN" as its value. "UP" means growing from lower to higher addresses. "DOWN" means growing from higher addressed to lower addresses.

- fill pattern

  The fill pattern attribute specifies with which pattern the stack area is initialized. This allows the debugger to evaluate the maximum stack usage.

  To provide system with information about application main stack area make the following is accomplished:

- for Diab complier: 2 variables are defined in Diab link command file: _OsOrtiStart (should be allocated at the same location as __SP_INIT Diab variable) and _OsOrtiStackStart (should be allocated at the same location as __SP_END Diab variable). In this case Diab link command file may look like the following:

```
__HEAP_START= ADDR(.bss)+SIZEOF(.bss);
__SP_INIT= ADDR(ram)+SIZEOF(ram);
__HEAP_END= __SP_INIT-SIZEOF(stack);
__SP_END= __HEAP_END;
```

For More Information: www.freescale.com

```
__DATA_RAM= ADDR(.data);
__DATA_ROM= __DATA_RAM;
__DATA_END= ADDR(.sdata)+SIZEOF(.sdata);
__BSS_START= ADDR(.sbss);
__BSS_END= ADDR(.bss)+SIZEOF(.bss);
/* 2 symbols for ORTI support */
_OsOrtiStart= __SP_INIT;
_OsOrtiStackStart= __SP_END;
```

- for Metrowerks CodeWarrior complier: 2 variables are defined in the OS code: _OsOrtiStart (contains the _stack_addr variable value) and _OsOrtiStackStart (contains the _stack_end variable value). These variables shall be pointed in the FORCEACTIVE statement in the link command file. In this case CodeWarrior link command file may look like the following:

```
MEMORY
{
...
}
FORCEACTIVE { _OsOrtiStart _OsOrtiStackStart}
SECTIONS
{
...
}
_heap_addr  = _e_bss;
_heap_end   = _e_bss+$(heapsize);
_stack_end  = _heap_end;
_stack_addr = _stack_end+$(stacksize);
```

The following counter information is available to the user:

- maxallowed value
- ticksperbase
- mincycle
- current counter value
- indicator if alarms attached to counter is activated
- property, which indicates OSTICKDURATION, the type of the counters (SWCOUNTER or HWCOUNTER) and timer hardware for system timers or indicates if it is an application counter

NOTE The current counter value is applicable for software counters only. The access to special purpose register is required for OS/MPC hardware counters, but this mechanism is not supported in ORTI.

The following alarm information is available to the user:

- alarm status
- assigned counter
- notified task and event
- time left to expire
- cycle value for cyclic alarm

NOTE The time left to expire is applicable for alarms attached to software counters only. The access to special purpose register is required for OS/MPC hardware counters, but this mechanism is not supported in ORTI.

The following resource information is available to the user:

- priority of the resource
- resource state

Depending on ORTI specification supported either message or message container information is generated. The following message information is available to the user:

- message type
- task to be notified and event
- callback function name
- notification flag

The message container describes each existing combination of messages and ACCESSORRECEIVED. The following message container information is available to the user:

- message name
- message type
- size of the queue
- number of valid messages in th equeue

For More Information: www.freescale.com

- first valid message
- message state
- action to be performed when message arrives
- message receiver
- message sender

Stack Debugging Support

Stack labels are provided to control OSEK OS stacks usage in OS/MPC for Diab These labels can be seen in symbol debugger and help control stack usage.

Stack labels

Stack labels define boundaries of the stack area for the Extended tasks. Bottom stack labels and top stack labels for tasks are provided.

Top of stack labels have the following format:

```
<task name>_TOS
```

Bottom of stack labels have the following format:

```
<task name>_BOS
```

Stack Overflow Checking

Optional stack overflow checking can be used during run time for check tasks, ISR and main stacks usage. Main and task stacks are checked during task switch. If stack overflow is detected the *ErrorHook* with E_OS_SYS_STACK, E_OS_SYS_MAINSTACK or E_OS_SYS_ISRSTACK status is called in case of *Extended Task* stack, Main stack or ISR stack overflow accordingly. Stack overflow control is turned on if *StackOverflowCheck* attribute value is TRUE and Error hook is defined. For main stack checking OS uses the labels defined in linker command file, *_stack_end* for Codewarrior and *__SP_END* for Diab compiler.

System checks task's and main stacks for overflow when tasks state is changed from *RUNNING* and before entering ISR of category 2. ISR stack (or main stack in case of BCC1) is checked before leaving ISR of category 2.

A

Sample Application

This appendix contains the text and listing of a sample customer application developed using OSEK OS.

This appendix consists of the following sections:

- Description
- Source Files

Description

The Sample application delivered with the OSEK Operating System should help to learn how to use OSEK OS. The Sample's source files are located in the SAMPLE directory – it contains all files needed to create an executable file.

The Sample is not a real application and it does not perform any useful work. But it has a certain algorithm so it is possible to track the execution. It uses most of OSEK OS mechanisms and allows the user to have the first look inside the OSEK OS.

The Sample consists of six tasks. It uses three counters (HW and SW are on the System Timer and Second Timer and one "user counter"), two alarms, TimeScale, two resources and two messages.

Generally, Sample tasks are divided into two parts. *TASKSND1, TASKSND2* and *TASKCNT* compose the first part (samplets.c file) and *TASKRCV1, TASKRCV2* and *TASKSTOP* are the second part (samplepc.c file). These two parts interact with the help of the messages and alarm mechanism.

The Extended task *TASKRCV1* is activated by OS (autostarted). It performs the following initializations:

- init *TSCOUNTER* counter with value 0,
- set absolute *STOPALARM* alarm to value 20 (when *STOPALARM* expired it activates *STOPTASK* task),

- starts *TimeScale*,
- clears *MSGAEVENT* and *TIMLIMITEVENT* events,
- set relative *TIMLIMITALARM* alarm to value 20
  (when *TIMLIMITALARM* expired it sets *TIMLIMITEVENT* event
  for this (*TASKRCV1*) task).

Then it enters *waiting* state - waiting *MSGAEVENT* and/or
TIMLIMITEVENT events. When event occurred, *TASKRCV1* checks
which event occurs.

If *MSGAEVENT* event occurred (*MsaA* message arrived) then
TASKRCV1 task:

- cancels *TIMLIMITEVENT* alarm,
- gets *MSGAACCESS* resource to prevent access to *MsgA* message,
- receives *MsgA* message with copy,
- releases *MSGAACCESS* resource,
- clears event and goes to waiting state again.

If *TIMLIMITEVENT* event occurred (time limit was exceeded) then
TASKRCV1 task goes to the very beginning and repeats
initialization, restarting all application (but OS is not restarted, it
continue running).

TimeScale has three steps:

1. On first step *TASKSND1* task is activated. It does the
   following:
- gets *MSGAACCESS* resource to prevent access to *MsgA* message,
- increments *MsgA* message value,
- send *MsgA* message without copy
  (*MsgA* message sets *MSGAEVENT* event for *TASKRCV1* task),
- releases *MSGAACCESS* resource,
- terminates itself.
2. On second step *TASKSND2* task is activated.
   Task *TASKSND2*:
- adds 3 to "ind" variable value,
- if "ind" variable value greater or equal 5 then subtracts 5 from
  "ind" value,

- stores *SYSTEMTIMER* value to body of *MsgB* message copy,

- gets *MSGBACCESS* resource to prevent access to *MsgB* message,

- send *MsgB* message with copy
  (message *MsgB* activates *TASKRCV2* task),

- releases *MSGBACCESS* resource,

- terminates itself.

3. On third step *TASKCNT* task is activated.
   TASKCNT task:

- increments *TSCOUNTER* counter
  (when *TSCOUNTER* counter value reaches 20 *STOPALARM*
  expires),

- terminates itself.

TASKRCV2 task:

- gets *MSGBACCESS* resource to prevent access to *MsgB* message,

- receives *MsgB* message without copy,

- releases *MSGBACCESS* resource,

- terminates itself.

TASKSTOP task:

- stops TimeScale (after a while *TIMLIMITALARM* expires)

- terminates itself.

The user can watch "ind" variable, messages content and so on.

Source Files

Source files for the Sample application are the following:

- samplets.c – the application code (*TASKSND1, TASKSND2*
  and *TASKCNT*)

- samplerv.c – the application code (*TASKRCV1, TASKRCV2*
  and *TASKSTOP*)

- sample.h – header file for the application code

- main.oil – OSEK Implementation Language file, platform
  independent part

- `cfg555cw.oil` – OSEK Implementation Language file for MPC555 platform for Metrowerks compiler
- `cfg555db.oil` – OSEK Implementation Language file for MPC555 platform for Diab Data compiler
- `cfg565cw.oil` – OSEK Implementation Language file for MPC565 platform for Metrowerks compiler
- `cfg565db.oil` – OSEK Implementation Language file for MPC565 platform for Diab Data compiler
- `cfg563cw.oil` – OSEK Implementation Language file for MPC561 platform for Metrowerks compiler
- `cfg563db.oil` – OSEK Implementation Language file for MPC561 platform for Diab Data compiler
- `msmake.bat` – command file for compiling sample using Microsoft nmake utility
- `gnumake.bat` – command file for compiling sample using GNU make utility

The directory structure of the Sample application is described in the readme.txt file located in the `sample\standard` directory.

To build the executable file the user should make sure that OSEKturbo OS components are properly installed on the disk and paths for the OSEK directory and compiler software are known. The makefiles were written for Microsoft Visual C++ 5.0 or v.6.0 nmake utility and for GNU make (from Cygwin package v.1.3.9) utility. Run the `MSMAKE.BAT` or `GNUMAKE.BAT` from DOS prompt to build the executable file.

To build sample application for CodeWarrior compiler run batch file with *codewarrior* parameter:

```
>MSMAKE codewarrior
```

To build sample application for Diab Data compiler run batch file with *diab* parameter:

```
>MSMAKE diab
```

By default, Sample application exception table is located at 0x00000000 address. Therefore correct MSR value (MSR IP bit is cleared) should be set prior to initial StartOS() call. If exception table

is located at 0xffffffff address, MSR IP bit should be set prior to initial StartOS() call and Sample makefile's should be adjusted accordingly (the address of "vect" section in the linker script). (See the "samplerv.c" file for detail).

Makefile uses the system environment variables DIABDIR and CWDIR to get compiler components, OSEKDIR to get OSEK components and SYSGENDIR to get SysGen shared components.

When all produced files are ready, the executable file can be load into the Evaluation board and run.

For More Information: www.freescale.com

B

System Service Timing

This appendix provides information about OS services execution time.

This appendix consists of the following sections:

- General Notes
- Data Sheet

General Notes

Results in tables below were got on the basis of the certain OS configuration. The list of system properties is shown below, and this configuration is called in the table as "Initial". Properties that are not listed have their default values. In the column "Configuration" the differences from the given list ("Initial") are indicated. For each configuration the corresponded numbers are provided in the table. In the column "Conditions" the specifics details for service execution are indicated.

```
CC = BCC1;
STATUS = STANDARD;
STARTUPHOOK = FALSE;
SHUTDOWNHOOK = FALSE;
PRETASKHOOK = FALSE;
POSTTASKHOOK = FALSE;
ERRORHOOK = FALSE;
USEGETSERVICEID = FALSE;
USEPARAMETERACCESS = FALSE;
FastTerminate = TRUE;
FastScheduler = TRUE;
ResourceScheduler = FALSE;
ACTIVATION = 1; (for all TASK objects)
SCHEDULE = FULL; (for all TASK objects)
ISR category 2 defined
Four tasks present in the application
```

Measurements were performed on ETAS EVB555 board. The numbers in the tables are the number of CPU system cycles counted by reading the free running time base counter in the columns titled "cycles" and in microseconds for 40MHzCPU

The "Latency" column presents the time for which the interrupts are disabled in the service.

OSEKturbo OS/MPC5xx application was compiled with compiler optimizations turned on and downloaded into onchip Flash memory, internal RAM was used for data and stack.

Measurements were done with the following conditions:

- Oscillator clock frequency = 4MHz
- TB tick duration = 1 µs (bit SCCR:TBS = 0)
- MCU frequency = 4 MHz (bits PLPRCR:MF = 0x003 and SCCR:DFNH = 2 to ensure 1µs TB tick duration)
- RCPU is not serialized (normal mode) and no show cycles will be performed for fetched instructions (bits ICTRL:ISCT_SER = 7)

It is possible that some real numbers can slightly differ from the presented values due to some last changes in OSEK OS.

Data Sheet

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
StartOS									
Initial		216	5.4	188	4.7	168	4.2	140	3.5

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
ActivateTask[a]									
Initial	Task activated, no rescheduling	52	1.3	20	0.5	52	1.3	20	0.5
	Task activated and rescheduled	80	2	40	1	80	2	44	1.1
Internal resource defined	Task activated and rescheduled	112	2.8	84	2.1	120	3	84	2.1
SCHEDULE = NON for all tasks	Task activated, no rescheduling	36	0.9	20	0.5	44	1.1	16	0.4
TerminateTask									
Initial	Task terminated, return to lower prio task	44	1.1	16	0.4	48	1.2	20	0.5
Initial	Task terminated, new task started	44	1.1	8	0.2	40	1	8	0.2
Internal resource defined	Task terminated, return to lower prio task	108	2.7	84	2.1	84	2.1	52	1.3
Internal resource defined	Task terminated, new task started	112	2.8	84	2.1	72	1.8	36	0.9
SCHEDULE = NON for all tasks	Task terminated, return to lower prio task	44	1.1	16	0.4	44	1.1	24	0.6
SCHEDULE = NON for all tasks	Task terminated, new task started	44	1.1	12	0.3	44	1.1	12	0.3

Freescale Semiconductor, Inc.

Table B.1   Run-time Services Timing Characteristics

| Configuration | Conditions | CodeWarrior | | | | Diab-CC | | | |
| | | Timing | | Latency | | Timing | | Latency | |
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
ChainTask[1]									
Initial	Task terminated, next task started	40	1	12	0.3	44	1.1	12	0.3
Internal resource defined	Task terminated, next task started	108	2.7	80	2	68	1.7	40	1
SCHEDULE = NON for all tasks	Task terminated, next task started	40	1	12	0.3	44	1.1	12	0.3
Schedule									
Initial	No rescheduling, all tasks are preemptable	32	0.8	4	0.1	32	0.8	8	0.2
Internal resource defined	No rescheduling, all tasks are preemptable	88	2.2	72	1.8	64	1.6	36	0.9
SCHEDULE = NON for all tasks	No rescheduling	28	0.7	8	0.2	28	0.7	8	0.2
	Rescheduling, other task becomes running	56	1.4	28	0.7	56	1.4	28	0.7
GetTaskId									
Initial		4	0.1	0	0	8	0.2	0	0
GetTaskState									
Initial	For running task	24	0.6	8	0.2	40	1	8	0.2
	For ready task	40	1	24	0.6	56	1.4	20	0.5
	For suspended task	36	0.9	24	0.6	48	1.2	20	0.5

For More Information: www.freescale.com

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
EnableAllInterrupts									
Initial		4	0.1	-	-	4	0.1	-	-
DisableAllInterrupts									
Initial		4	0.1	-	-	4	0.1	-	-
ResumeAllInterrupts									
Initial		24	0.6	-	-	28	0.7	-	-
SuspendAllInterrupts									
Initial		20	0.5	-	-	24	0.6	-	-
ResumeOSInterrupts									
Initial		28	0.7	-	-	28	0.7	-	-
SuspendOSInterrupts									
Initial		20	0.5	-	-	28	0.7	-	-
GetResource									
a resource defined	Task occupies resource	56	1.4	20	0.5	56	1.4	24	0.6
call from ISR level	ISR occupies resource	56	14	-	-	56	1.4	-	-
ReleaseResource									
a resource defined	Task releases resource, no rescheduling	76	1.9	40	1	72	1.8	36	0.9
	Task releases resource, with rescheduling	112	2.8	72	1.8	100	2.5	68	1.7
call from ISR level	ISR releases resource	68	1.7	-	-	68	1.7	-	-

Freescale Semiconductor, Inc.

System Service Timing
Data Sheet

Table B.1 Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
SetEvent[1]									
CC = ECC1 SCHEDULE = NON for all tasks an event defined	Event set, but task was not waiting it	60	1.5	24	0.6	64	1.6	24	0.6
	Event set, task becomes ready	72	1.8	32	0.8	76	1.9	32	0.8
CC = ECC1 an event defined	Event set, but task was not waiting it	64	1.6	24	0.6	84	2.1	40	1
	Event set, task becomes ready	84	2.1	40	1	100	2.5	52	1.3
	Event set, task becomes running	212	5.3	180	4.5	228	5.7	184	4.6
ClearEvent									
CC = ECC1 SCHEDULE = NON for all tasks an event defined		20	0.5	8	0.2	36	0.9	4	0.1
CC = ECC1 an event defined		24	0.6	4	0.1	36	0.9	4	0.1
GetEvent									
CC = ECC1 SCHEDULE = NON for all tasks an event defined		12	0.3	0	0	12	0.3	0	0
CC = ECC1 an event defined		12	0.3	0	0	8	0.2	0	0

TR–272 OSEKturbo OS/MPC5xx

For More Information: www.freescale.com

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
WaitEvent									
CC = ECC1 SCHEDULE = NON for all tasks an event defined	Event was already set, task remains running	28	0.7	8	0.2	36	0.9	8	0.2
	Task becomes waiting	164	4.1	144	3.6	168	4.2	140	3.5
CC = ECC1 an event defined	Event was already set, task remains running	28	0.7	0	0	36	0.9	8	0.2
	Task becomes waiting	172	4.3	140	3.5	168	4.2	140	3.5
SendMessage									
an unqueued message with copy	Message size 16 bytes	188	4.7	24	0.6	124	3.1	20	0.5
ReceiveMessage									
an unqueued message with copy	Message size 16 bytes	188	4.7	28	0.7	124	3.1	24	0.6
InitCounter									
a counter defined		8	0.2	0	0	12	0.3	4	0.1
CounterTrigger									
a counter defined		68	1.7	36	0.9	60	1.5	32	0.8
1 alarm is set to a counter	Alarm is not expired	128	3.2	96	2.4	100	2.5	68	1.7
	Notified task becomes ready	160	4	124	3.1	132	3.3	104	2.6
	Notified task becomes running	188	4.7	152	3.8	160	4	132	3.3

Freescale Semiconductor, Inc.

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
10 tasks, 10 alarms are set to a counter	Alarm is not expired	292	7.3	268	6.7	204	5.1	184	4.6
	Notified tasks become ready	632	15.8	596	14.9	568	14.2	544	13.6
	High priority notified task becomes running	664	16.6	624	15.6	596	14.9	572	14.3
GetCounterValue									
a counter defined		16	0.4	0	0	12	0.3	0	0
GetCounterInfo									
a counter defined		24	0.6	0	0	24	0.6	8	0.2
GetAlarmBase									
an alarm defined		40	1	0	0	24	0.6	4	0.1
SetRelAlarm									
an alarm defined	Alarm set	136	3.4	96	2.4	124	3.1	80	2
	Alarm expires immediately, notified task becomes ready[1]	236	5.9	196	4.9	200	5	152	3.8
	Alarm expires immediately, notified task becomes running	260	6.5	208	5.2	216	5.4	172	4.3

For More Information: www.freescale.com

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
10 alarm are defined, 9 alarms are set on a counter	Alarm set	132	3.3	100	2.5	128	3.2	76	1.9
	Alarm expires immediately, notified task becomes ready[1]	404	10.1	372	9.3	308	7.7	264	6.6
	Alarm expires immediately, notified task becomes running	428	10.7	388	9.7	328	8.2	288	7.2
SetAbsAlarm									
an alarm defined	Alarm set	104	2.6	68	1.7	96	2.4	56	1.4
	Alarm expires immediately, notified task becomes ready[1]	204	5.1	168	4.2	172	4.3	128	3.2
	Alarm expires immediately, notified task becomes running	228	5.7	180	4.5	192	4.8	148	3.7
10 alarm are defined, 9 alarms are set on a counter	Alarm set	104	2.6	68	1.7	96	2.4	52	1.3
	Alarm expires immediately, notified task becomes ready[1]	372	9.3	340	8.5	276	6.9	240	6
	Alarm expires immediately, notified task becomes running	396	9.9	360	9	300	7.5	264	6.6
CancelAlarm									
1 alarm is set on a counter		40	1	20	0.5	52	1.3	20	0.5
10 tasks, 10 alarms are set on a counter		36	0.9	24	0.6	52	1.3	24	0.6

OSEKturbo OS/MPC5xx   **TR−275**

Table B.1   Run-time Services Timing Characteristics

Configuration	Conditions	CodeWarrior				Diab-CC			
		Timing		Latency		Timing		Latency	
		cycles	µs	cycles	µs	cycles	µs	cycles	µs
GetAlarm									
1 alarm is set on a counter		52	1.3	20	0.5	60	1.5	20	0.5
10 tasks, 10 alarms are set on a counter		44	1.1	20	0.5	56	1.4	20	0.5
StartTimeScale									
TimeScale defined	Task activated, no rescheduling[1]	144	3.6	120	3	132	3.3	100	2.5
	Task activated, and rescheduled	172	4.3	144	3.6	164	4.1	128	3.2
StopTimeScale									
TimeScale defined		40	1	0	0	44	1.1	0	0

[a]. The execution time of this service depends on the number of ready tasks of the priority higher than activated task

The table below contains interrupt frame characteristics for OSEKturbo OS/MPC5xx, and Diab-DCC and Metrowerks compilers.

Table B.2   Interrupt Frame Characteristics for ISR Category 2[a]

Configuration	Conditions	DIAB-DCC		Metrowerks	
		cycles	µs	cycles	µs
From interrupt until first ISR command					
Initial		76	1.9	72	1.8

For More Information: www.freescale.com

Table B.2   Interrupt Frame Characteristics for ISR Category 2[a]

Configuration	Conditions	DIAB-DCC		Metrowerks	
		cycles	µs	cycles	µs
From last ISR command until task/idle loop					
Initial	Return to interrupted task	60	1.5	60	1.5
	Task rescheuling occurs	64	1.6	60	1.5
	Return to Idle Loop	60	1.5	56	1.4

a. Interrupt frame characteristics are the data about delay from interrupt until start of first ISR command and from end of last ISR command to first instruction of task/idl loop for ISR category 2.

C

Memory Requirements

This appendix provides information about the amount of ROM and RAM directly used by various versions of the OSEK OS.

This appendix consists of the following sections:

- Memory for the OSEK Operating System
- Data Sheet for Diab-CC
- Data Sheet for CodeWarrior

Memory for the OSEK Operating System

The table below contains the data about ROM and RAM needed for the OSEK Operating System kernel and system objects. The amount of memory depends on the system configuration and on the number of certain objects (e.g., tasks, counters, etc.). The table does not reflects all possible configurations so the overall number of them is too big. Therefore, only some most important configurations are presented.

The following initial system property settings were used to determine memory requirements:

```
CC = BCC1;
STATUS = STANDARD;
STARTUPHOOK = FALSE;
SHUTDOWNHOOK = FALSE;
PRETASKHOOK = FALSE;
POSTTASKHOOK = FALSE;
ERRORHOOK = FALSE;
USEGETSERVICEID = FALSE;
USEPARAMETERACCESS = FALSE;
ResourceScheduler = FALSE;
FastTerminate = TRUE;
FastScheduler = TRUE;
MessageCopyAllocation = USER;
ACTIVATION = 1; (for all TASK objects)
```

```
SCHEDULER = FULL; (for all TASK objects)
ISR category 2 not defined
```

This initial property list was used for the first row in the table. It conforms to the BCC1 Conformance Class without any additional mechanisms and this is the minimal OSEK OS configuration. The rows below reflects memory requirements for the next Conformance Classes. System properties are shown in the rows which are turned on for the corresponded Conformance Class.

All other rows below the first one ("Initial") has a title "<u>Initial</u>" or "<u>Changed:</u>" and one or more options turned ON or OFF. If a row has a title "Initial" it means that for such OS configuration the Initial property list is used with particular options changed as shown. If a row has a title "<u>Changed:</u>" it means that for such OS configuration the setting list as for the previous row is used with particular options changed as shown. Thus, the system functionality grows up.

Since each system object (a task, a message, an alarm, etc.) requires some ROM and RAM the total amount of memory depends on the number of objects. Therefore, the formulas should be used to calculate the exact memory amount for each case. These formulas are provided in the table.

Data presented in the table do not include ISR stacks for RAM and the MCU startup code, task functions and the vector table for ROM.

In the formulas in the table the following symbols are used:

- T is the number of Tasks
- P is the number of priorities (Tasks+Resources)
- R is the number of Resources (without RES_SCHEDULER)
- IR is the number of Resources with ISR priority
- C is the number of Counters
- A is the number of Alarms
- M is the number of Messages

Data Sheet for Diab-CC

Table C.1   Operating System Memory Requirements (Diab-CC)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
0	Initial	BCC1	13+1*T	820+5*T
1	**Initial** SCHEDULE = NON for all tasks		13+1*T	756+5*T
2	**Initial** SCHEDULE = NON for one task and FULL for other one		13+1*T	836+5*T
3	**Changed**: STATUS = EXTENDED		101+88*T	1356+8*T
4	**Changed**: ERRORHOOK = TRUE		102+88*T	1744+8*T
5	**Changed**: USEGETSERVICEID = TRUE USEPARAMETERACCESS = TRUE		107+88*T	1984+8*T
6	**Initial**: Counter defined		13+1*T+8*C	1076+5*T+8*C
7	**Changed**: Alarm with ACTION=ACTIVATETASK defined		13+1*T+12*C+20*A	1932+5*T+8*C+2*A
8	**Initial**: Message with ACTION=NONE defined		13+1*T+16*M	1224+5*T+12*M
9	**Initial**: Message with ACTION=ACTIVATETASK defined		13+1*T+20*M	1372+5*T+16*M
10	**Initial**: Message with ACTION=CALLBACK defined		13+1*T+20*M	1312+5*T+16*M
11	**Initial**: Resource for task defined		113+92*T	1200+8*T+1*R
12	**Changed**: ISR category 2 defined		114+92*T	1876+8*T+1*R
13	**Changed**: InterruptDispatcher = OneLevel		130+92*T	2200+8*T+1*R
14	**Changed**: Resource referenced by ISR		138+92*T	2372+8*T+1*R

Table C.1   Operating System Memory Requirements (Diab-CC)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
15	Initial:	ECC1	105+92*T	1324+12*T
16	**Initial**: an event defined		113+100*T	1816+12*T
17	**Changed** SCHEDULE = NON for all tasks		113+100*T	1688+12*T
18	**Initial** an event defined, SCHEDULE = NON for one task and FULL for other one		113+100*T	1840+12*T
19	**Changed**: STATUS = EXTENDED		113+100*T	2028+12*T
20	**Changed**: ERRORHOOK = TRUE		114+100*T	2784+12*T
21	**Changed**: USEGETSERVICEID = TRUE USEPARAMETERACCESS = TRUE		119+100*T	3324+12*T
22	**Initial**: an event defined, Counter defined		113+100*T+8*C	2076+12*T+8*C
23	**Changed**: Alarm with ACTION=SETEVENT defined		113+100*T+12*C+24*A	3112+12*T+8*C+12*A
24	**Initial**: an event defined, Message with ACTION=NONE defined		113+100*T+16*M	2220+12*T+12*M
25	**Initial**: an event defined, Message with ACTION=ACTIVATETASK defined		113+100*T+20*M	2412+12*T+16*M
26	**Initial**: an event defined, Message with ACTION=SETEVENT defined		113+100*T+24*M	2356+12*T+20*M
27	**Initial**: an event defined, Resource for task defined		125+104*T	2124+12*T+1*R
28	**Changed**: ISR category 2 defined		126+104*T	3108+12*T+1*R
29	**Changed**: InterruptDispatcher = OneLevel		142+104*T	3432+12*T+1*R

Table C.1   Operating System Memory Requirements (Diab-CC)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
30	**Changed**: Resource referenced by ISR	ECC1	130+104*T	3280+12*T+1*R

For More Information: www.freescale.com

Data Sheet for CodeWarrior

Table C.2   Operating System Memory Requirements (CodeWarrior)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
0	Initial	BCC1	13+1*T	624+5*T
1	**Initial** SCHEDULE = NON for all tasks		13+1*T	584+5*T
2	**Initial** SCHEDULE = NON for one task and FULL for other one		13+1*T	640+5*T
3	**Changed**: STATUS = EXTENDED		101+88*T	1300+8*T
4	**Changed**: ERRORHOOK = TRUE		102+88*T	1712+8*T
5	**Changed**: USEGETSERVICEID = TRUE USEPARAMETERACCESS = TRUE		107+88*T	1916+8*T
6	**Initial**: Counter defined		13+1*T+8*C	880+5*T+8*C
7	**Changed**: Alarm with ACTION=ACTIVATETASK defined		13+1*T+12*C+20*A	1840+5*T+8*C+2*A
8	**Initial**: Message with ACTION=NONE defined		13+1*T+16*M	1004+5*T+12*M
9	**Initial**: Message with ACTION=ACTIVATETASK defined		13+1*T+20*M	1168+5*T+16*M
10	**Initial**: Message with ACTION=CALLBACK defined		13+1*T+20*M	1140+5*T+16*M
11	**Initial**: Resource for task defined		105+92*T+8*R	1152+8*T+1*R
12	**Changed**: ISR category 2 defined		106+92*T+8*R	1616+8*T+1*R
13	**Changed**: InterruptDispatcher = OneLevel		122+92*T+8*R	1940+8*T+1*R
14	**Changed**: Resource referenced by ISR		126+92*T+12*R	2136+8*T+1*R

For More Information: www.freescale.com

Table C.2   Operating System Memory Requirements (CodeWarrior)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
15	Initial:	ECC1	105+92*T	1284+12*T
16	**Initial**: an event defined		113+100*T	1800+12*T
17	**Changed** SCHEDULE = NON for all tasks		113+100*T	1732+12*T
18	**Initial** an event defined, SCHEDULE = NON for one task and FULL for other one		113+100*T	1824+12*T
19	**Changed**: STATUS = EXTENDED		113+100*T	1996+12*T
20	**Changed**: ERRORHOOK = TRUE		114+100*T	2776+12*T
21	**Changed**: USEGETSERVICEID = TRUE USEPARAMETERACCESS = TRUE		119+100*T	3244+12*T
22	**Initial**: an event defined, Counter defined		113+100*T+8*C	2060+12*T+8*C
23	**Changed**: Alarm with ACTION=SETEVENT defined		113+100*T+12*C+24*A	3160+12*T+8*C+12*A
24	**Initial**: an event defined, Message with ACTION=NONE defined		113+100*T+16*M	2180+12*T+12*M
25	**Initial**: an event defined, Message with ACTION=ACTIVATETASK defined		113+100*T+20*M	2384+12*T+16*M
26	**Initial**: an event defined, Message with ACTION=SETEVENT defined		113+100*T+24*M	2360+12*T+20*M
27	**Initial**: an event defined, Resource for task defined		117+104*T+8*R	2140+12*T+1*R
28	**Changed**: ISR category 2 defined		118+104*T+8*R	2872+12*T+1*R
29	**Changed**: InterruptDispatcher = OneLevel		134+104*T+8*R	3196+12*T+1*R

Table C.2   Operating System Memory Requirements (CodeWarrior)

Num-ber	System Properties (configuration)	Confor-mance Class	RAM	ROM
30	**Changed**: Resource referenced by ISR	ECC1	122+104*T+12*R	3068+12*T+1*R

For More Information: www.freescale.com

D

System Generation Error Messages

This appendix explains OSEK OS System Generator error messages.

The System Generator checks the compatibility of properties, parameters and limits and reports about possible errors via error messages. The error messages can be associated with the wrong syntax, mistakes in the implementation definition, wrong definitions of the application objects.

This appendix consists of the following sections:

- Severity Level
- Error Message Format
- List of Messages

Severity Level

The messages vary in their severity level, they can be one of the following types: ***information***, ***warning***, ***error***, ***fatal error***. Usually an information message attends other type of error message and contains reference to necessary information associated with error situation. A warning message only prevents about possible error. If an error message is detected, than the operation that should be started after the current one, is will not be executed. For example, if the error messages were found in project verification, the configuration file will not be generated however project settings check will be continued. When a fatal error message is found, than anyone of build command is terminated.

NOTE All warning messages or some types of them can be suppressed by using the **-w** option.

Error Message Format

The error message format depends on mode in which SysGen has been running. By default an error message includes the file name, the line number, the error code and a short error description. The error messages have one of the following formats in accordance with the severity level of message:

```
[<filename>(<line_number>) : ]information <prefix>####:<message>
[<filename>(<line_number>) : ]warning <prefix>####:<message>
[<filename>(<line_number>) : ]error <prefix>####:<message>
[<filename>(<line_number>) : ]Fatal Error <prefix>####:<message>
```

where:

`<filename>`	file name
`<line_number>`	line number
`<prefix>`	component specific prefix
`####`	number of message
`<message>`	short description of the error

The message format can be set to a tree-like form (if possible, for example if an error is encountered while the project verifying) by means -*b* option. The message includes the object type, the object name, the error code and a short error description. In this case the error message has the following format:

```
CPU<name>,<OBJECT><name>,<attribute>:information<prefix>####:<message>
CPU<name>,<OBJECT><name>,<attribute>:warning<prefix>####:<message>
CPU<name>,<OBJECT><name>,<attribute>:error<prefix>####:<message>
CPU<name>,<OBJECT><name>,<attribute>:Fatal Error<prefix>####:<message>
```

where:

`<name>`	name of corresponding object - CPU or object
`<OBJECT>`	keyword used to identify object type in OIL
`<attribute>`	name of outermost attribute (in some cases all nested attributes are presented through dots). Optional, present, if the message is associated with a particular attribute

`<prefix>`	component specific prefix
`####`	number of message
`<message>`	short description of the error

Below error and warning messages generated by SysGen components are described.

List of Messages

SysGen consists of several components, which work on different stages of OIL file processing. Each component generates messages with specific prefix before the number. The following prefixes are used: **SG** - for SysGen Engine main component, **OR** - for OIL Reader component, which reads specified OIL file into internal repository, **TD** - for Target-Specific DLL, which process stored OIL file with target platform awareness.

SysGen Engine Messages

SG0001: **Option '-v' is used, ignoring other options**
Information

When **-v** option is defined, System Generator is run only to output versions of all its components. So all other options are ignored.

SG0002: **Option <option> has been redefined; effective value is <value>**
Warning

Only one value can be used for the following options: **-c**, **-h**, **-n**, **-o**, **-p**. Being defined twice, their last value will be used as SysGen argument.

SG0003: **Option <option> requires argument**
Error

The following options should be defined with an argument: **-c**, **-f**, **-h**, **-i**, **-n**, **-o**, **-p**, **-s**.

SG0004: **Option <option> does not accept arguments**
Error

The following options should not be defined with an argument: **-b**, **-t**, **-v**.

SG0005: **Unexpected token <token>**

Error

An unrecognizable token was found in the input stream. The SysGen option or OIL file name is expected.

SG0006: **Closing quotation mark is expected**

Fatal Error

The quotation mark is missed. The command line contains odd number of quotes.

SG0007: **Cannot open command argument file**

Fatal Error

The command argument file cannot be open if it does not exist or you do not have permission to open the file or directory.

SG0008: **Cyclic reference in the command argument file**

Fatal Error

The command file contains **-f** option that has a reference to the same file as an argument. Also some command files can have number of recursive references.

SG0009: **Input OIL file must be defined**

Fatal Error

The input OIL file cannot be found if it does not exist or you do not have permission to open the file or directory.

SG0010: **Processing halted; possibly there is not enough memory**

Fatal Error

The message is generated by the SysGen either there is no enough free RAM memory on users' PC or some internal error is occurred and it would be resolved with the help of support team.

Please, report to osek@helpline.sps.mot.com.

SG0201: **Error loading Target DLL <name>**

Fatal Error

Target DLL with the specified name is absent.

SG0202: **Wrong Target DLL <name> version**

Fatal Error

Target DLL version shall be 1.x or 2.x. Other versions of Target DLL are not supported.

SG0203: **Target DLL <name> failed**

Fatal Error

Target DLL is corrupted.

SG0204: **Target DLL <name> cannot be found**

Fatal Error

The Target DLL name is not found in the Windows Registry.

SG0301: **Implementation definition database in Windows Registry is invalid or does not exist**

Fatal Error

Implementation definition database defined by installation program, was corrupted or removed.

SG0303: **Container <name> does not exist**

Fatal Error

Implementation definition database does not contain information about specified container.

SG0401: **Wrong license**

Fatal Error

The error message is generated if OSEKTURBO or FULL feature is not defined in the OSEK OS license. Also license file can be corrupted. For more detailed information see Flex License Manager documentation.

SG5001: **Cannot open template file <filename>**

Fatal Error

The System Generator cannot open template file for use if the template file which name defined in the Windows Registry, does not exist or you do not have permission to open the file or directory.

SG5002: **Cannot open output file <filename>**

Fatal Error

The output file cannot be opened if there is no enough free disk space or there is no appropriate permission to create or write the output file.

SG5003: **Template file <filename> is corrupt or invalid**

Fatal Error

The contents of the template file are corrupt.

SG5004: **Cannot write output file <filename>. Possibly not enough space on drive**

Fatal Error

The output file cannot be written if there is no enough free disk space or there is no appropriate permission to write the output file.

OIL Reader Messages

OR0000: **Cannot open input file**

Fatal Error

The file does not exist, or you do not have permission to open the file or directory.

OR0002: **Unterminated file path**

Fatal Error

The incorrect syntax of an include directive.

OR0003: **<file path> or "file path" expected**

Fatal Error

The incorrect syntax of an include directive.

OR0004: **Wrong #include directive format**

Fatal Error

The incorrect syntax of an include directive.

OR0053: **OIL version expected**

Fatal Error

OIL version should be defined as value for the *OIL_VERSION* attribute at the beginning of OIL file.

OR0005: **Unclosed comment**

Error

The OIL file may contain C++-style comments (/* */ and //). If the comment is started be /*, then the end of the comment is defined by */.

OR0006: **Unknown directive**

Error

Unknown preprocessing directive was detected.

OR0007: **Unterminated string**

Error

The attribute's value of STRING type should be completed by double-quote.

OR0008: **Wrong number format**

Error

The number format on any unsigned integer number (possibly restricted to a range of numbers).

OR0009: **Floating point is not supported**

Error

Floating point is not supported by the 2.0 and 2.0e OIL versions. The message is generated if an OIL file of 2.0 or 2.0e format contains an attribute of FLOAT type.

OR0010: **Syntax error**

Error

A syntax error was found in the input stream. Problems of this type can sometimes be attributed to a syntactical or clerical error. For example:

```
TASK taskA
{
    PRIORITY = 5    // No closing semicolon
    SCHEDULE = FULL;
};
```

In the preceding example, the error message will be generated for the line which contains the definition of *SCHEDULE* attribute, although the true source of the error appears on the line just above. As a general rule, make sure to also examine the lines above the line listed in the error message when trying to determine the cause.

OR0011: Attribute value does not match type

Error

The value does not correspond to attribute type.

OR0012: Attribute value is out of range

Error

The attribute value is greater or less than range defined.

OR0013: Attribute range does not match type

Error

The value range does not correspond to attribute type.

OR0014: Range bounds are wrong

Error

The first bound of the value range defines the minimal value of the attribute, and the second bound specifies the maximal value, i.e. the first bound has to be less than second one. For example, [1..256].

OR0015: Range is not allowed

Error

Object attributes of some types (ENUM, BOOLEAN, STRING) and object references have not to be defined with value range in the implementation definition part.

OR0016: ENUM attribute requires value list

Error

The attributes of ENUM type have to be defined with one or more values allowed for the attribute.

OR0017: AUTO is not allowed

Error

If the implementation definition part has an attribute with the WITH_AUTO specifier, then the AUTO value can be set for the attribute.

OR0018: Duplicated values in ENUM attribute

Error

Values listed for an attribute of ENUM type, have to differ each other.

OR0019: Range of BOOLEAN attribute must be [TRUE, FALSE]

Error

The attribute of BOOLEAN type can have either TRUE or FALSE value.

OR0020: Reference type must be appended with "_TYPE"

Error

The reference type is taken from the referenced object and _TYPE particle, e.g. a reference to a task shall use the TASK_TYPE keyword as reference type.

OR0021: Attribute type doesn't match the type of one previously defined

Error

The same subattributes cannot be defined with different types.

OR0022: Duplicated attribute definition

Error

The implementation part of OIL file contains two definition of the same attribute.

OR0023: Wrong reference type

Error

According to OIL standard the reference type is taken from the referenced object and _TYPE particle, e.g. a reference to a task shall use the TASK_TYPE keyword as reference type.

OR0024: Duplicated predefined attribute

Error

The predefined attributes *CONTAINER* and *SECTION* are already defined (for OIL2.0e).

OR0025: **Standard object <name> is not defined**

Error

The implementation definition must contain all standard objects.

OR0026: **Standard attribute <name> is not defined**

Error

The implementation definition must contain all standard attributes.

OR0027: **Standard attribute must have WITH_AUTO specifier**

Error

This attribute definition does not correspond to OIL standard.

OR0028: **Standard attribute must have [ ] specifier**

Error

This attribute definition does not correspond to OIL standard.

OR0029: **Standard attribute must not have [ ] specifier**

Error

This attribute definition does not correspond to OIL standard.

OR0030: **Standard attribute must have range**

Error

This attribute definition does not correspond to OIL standard.

OR0031: **Standard attribute range cannot be expanded**

Error

This attribute definition does not correspond to OIL standard. The value range for standard attributes may be restricted but not expanded.

OR0032: **Standard attribute must have ENUM value list**

Error

This attribute definition does not correspond to OIL standard.

OR0033: **Standard dependent attributes are expected**

Error

According to OIL standard the attribute has to contain subattributes' definition.

OR0034: **Standard attribute ENUM value list cannot be expanded**

Error

This attribute definition does not correspond to OIL standard. The value range of standard attributes may be restricted but not expanded.

OR0035: **Standard reference type mismatch**

Error

The type of standard reference does not correspond to OIL standard.

OR0036: **Standard attribute type mismatch**

Error

The type of standard attribute does not correspond to OIL standard.

OR0037: **Duplicated object**

Error

The object has been already defined in the implementation definition.

OR0038: **CFG <name> is not defined**

Error

CFG statements are not supported by OIL standard.

OR0039: **No implementation is found**

Error

No implementation definition was found in the input file.

OR0040: **Container type does not match implementation**

Error

The OIL file contains a container type that is not defined by implementation.

OR0041: **Section name does not match implementation**

Error

The OIL file contains a section type that is not defined by implementation.

OR0042: **Type of object is not defined in the implementation**

Error

Only standard objects defined by OIL specification shall be used in application definition. New object types are not allowed.

OR0043: **Attribute not defined in the implementation**

Error

This attribute is not specified in the implementation definition.

OR0044: **Attribute is already defined**

Error

The attribute with specified name has been already defined.

OR0045: **Single value is expected**

Error

The attribute requires single value (for OIL 2.0 and 2.0e).

OR0046: **Dependent attributes corresponded to wrong value not allowed**

Error

The subattributes' values in the application do not correspond to them in the implementation definition.

OR0047: **Reference is expected**

Error

Invalid name of the referenced object.

OR0048: **Referenced object is not found**

Error

The referenced object with specified name is not found.

OR0049: **Reference type mismatch**

Error

The referenced object type does not correspond to reference type.

OR0050: **Container attributes are not allowed**

Error

The message is generated if CPU container is defined by attributes. Container attributes are allowed only for OIL 2.0e format.

OR0051: Attribute <name> must be defined

Error

The standard parameters and attributes with NO_DEFAULT specifier
defined in implementation definition must be defined in the application
definition (except references).

**OR0052: Dependent NO_DEFAULT attribute is defined in the default branch of
his parent**

Error

The implementation definition must not have attributes with the default
value that contain subattributes with NO_DEFAULT specifier.

OR0054: Standard attribute must not have WITH_AUTO specifier

Error

This attribute definition does not correspond to OIL standard.

OR0056: Object is already defined with different type

Error

The given name has already been used for a system object of the other type.

OR0057: <type> object must be defined

Error

The OS or APPMODE object cannot be missed in OIL file.

OR0058: Only one <type> object can be defined

Error

Only one OS object can be defined.

OR0059: Implementation is already defined

Error

Only one implementation can be defined in OIL file.

OR0060: CPU is already defined

Error

Only one CPU container can be defined in OIL file.

OR0061: No CPU in the file

Error

CPU container must be defined in OIL file.

OR0062: Include directive within object is not allowed

Error

The #include directive(s) cannot be placed inside object definition.

OR0063: Related item location

Information

The message refers to the item in implementation or application definition that is relevant to previous error message.

OR0064: Only CPU container allowed

Error

Only CPU container is supported by OIL format.

OR0065: Nested objects are not allowed

Error

The message is generated when OIL file is read. Subobjects cannot be defined in the object definition according to OIL standard.

OR0066: Unsupported OIL Version

Fatal Error

OSEK SysGen supports OIL 2.1, OIL 2.2 and OIL2.3.

OIL version should be defined as value for the OIL_VERSION attribute at the beginning of OIL file.

OR0067: <character> expected

Information

This character is expected according to OIL file syntax.

OR0068: String expected

Information

The string is expected according to OIL file syntax.

OR0069: Implementation name expected

Information

The implementation name has to be defined according to OIL file syntax.

OR0070: Value expected

Information

The attribute value has to be defined according to OIL file syntax.

OR0071: Attribute name expected

Information

The attribute name has to be defined according to OIL file syntax.

OR0072: Unknown character <char>

Information

This character has not to be used in the OIL file. The symbol code is presented.

OR0073: Cyclic #include reference

Information

Cyclic #include references are not allowed.

OR0074: Object <type> is not standard

Warning

Nonstandard object types have not to be defined in the implementation definition.

OR5011: Unknown object in implementation part

Error

The object defined in OIL file absents in the corresponding implementation definition file.

OR5012: Unknown attribute in implementation part

Error

The attribute defined in OIL file absents in the corresponding implementation definition file.

OR5013: Type of attribute is invalid

Error

The attribute type defined in OIL file does not correspond to the registered implementation definition file.

OR5014: Type of reference is invalid

Error

The attribute type defined in OIL file does not correspond to the registered implementation definition file.

OR5017: Single value <name> attribute is declared as multiple value

Error

The attribute defined in OIL file as multiple has single value in the registered implementation definition file.

OR5018: Multiple value <name> attribute is declared as single value

Error

The attribute defined in OIL file as single has multiple value in the registered implementation definition file.

OR5019: Attribute must not have WITH_AUTO specifier

Error

The attribute is defined in OIL file with WITH_AUTO specifier, but this is inconsistent with the corresponding implementation definition file.

OR5020: Attribute must have WITH_AUTO specifier

Error

The attribute is defined in OIL file without WITH_AUTO specifier, but this is inconsistent with the corresponding implementation definition file.

OR5021: Attribute must not have default value

Error

The default value is defined for the attribute in OIL file, but the corresponding implementation definition file does not specify default value for this attribute.

OR5022: **Wrong attribute default value**

Error

The attribute default value defined in OIL file does not correspond to it in the appropriate implementation definition file.

OR5023: **Attribute must have default value**

Error

The default value is not defined for the attribute in OIL file, but the corresponding implementation definition file specifies the default value for this attribute.

OR5024: **Attribute must not have range**

Error

The value range is defined for the attribute in OIL file, but the corresponding implementation definition file does not assign default value for this attribute.

OR5025: **Wrong minimum value**

Error

The minimum bound in the value range defined for the attribute in OIL file and in the corresponding implementation definition file is inconsistent.

OR5026: **Wrong maximum value**

Error

The maximum bound in the value range defined for the attribute in OIL file and in the corresponding implementation definition file is inconsistent.

OR5027: **Attribute must have range**

Error

The value range defined for the attribute in the implementation definition file, is absent for this attribute in the appropriate OIL file.

OR5028: **Wrong ENUM value**

Error

The value list defined for the attribute in OIL file and in the corresponding implementation definition file is inconsistent.

OR5029: Some ENUM values are not defined for attribute

Error

Some of the values defined for the attribute in the implementation definition file, is absent for this attribute in the appropriate OIL file.

OR5030: Unknown container type

Error

The application definition specifies a container type that was not defined.

OR5031: Unknown section

Error

The application definition specifies a section that was not defined (for OIL 2.0e).

OR5032: Unknown implementation

Error

The application definition specifies an implementation that was not defined (for OIL 2.0e).

OR5050: Attribute must not have NO_DEFAULT value

Error

The NO_DEFAULT specifier is defined for the attribute in OIL file, but in the corresponding implementation this attribute does not have this specifier.

OR5051: Attribute must have NO_DEFAULT value

Error

The NO_DEFAULT specifier is not defined for the attribute in OIL file, but the corresponding implementation definition file specify it for this attribute.

OR5060: Object not defined

Error

The object is defined in the implementation definition file, but it is absent in the appropriate OIL file.

OR5061: Attribute not defined

Error

The attribute is defined in the implementation definition file, but it is absent in the appropriate OIL file.

OR5062: Depended attribute group for value: <value> is absent

Error

The values of subattributes shall be defined if these parameters do not have default values in the implementation definition.

OR5101: Can't create mode <name>, ignored

Warning

The message is generated when OIL file was converted from 2.0e OIL format to 2.1 format. Application mode in 2.1 OIL format contains only references to tasks defined in the application. APPMODE object is not created.

Target-Specific DLL Messages

TD0001: Wrong license

Fatal Error

The error message is generated if OSEKTURBO feature is not defined in the OSEK OS license.

TD0006: EVENTs are not allowed for <class>

Error

According to OSEK OS spec. the events are not supported in Basic conformance classes. This message is generate if the EVENT object is defined, but one of the Basic conformance classes is defined for OS.

TD0007: EVENT is defined in <class>

Information

According to OSEK OS spec. the events are not supported in Basic conformance classes. This message is generate if the EVENT object is defined, but one of the Basic conformance classes is defined for OS.

TD0023: IsrStackSize shall be defined

Error

The *IsrStackSize* attribute must be defined if the application contains extended tasks and ISR category 2 or system timer.

TD0038: **At least one OS/TASK/APPMODE shall be defined**

Error

At least one object of OS, TASK, and APPMODE types should to be defined in an application. Only one OS object has to be defined.

TD0045: **COUNTER for SysTimer / SecondTimer shall be defined**

Error

If the *SysTImer/SecondTimer* attribute is SWCOUNTER or HWCOUNTER, then the *COUNTER* reference shall be defined. The same counter cannot be attached to both system and second timers.

TD0046: **Same COUNTER cannot be used for SysTimer and SecondTimer**

Error

The same counter cannot be attached to both system and second timers.

TD0051: **Extended TASK is not supported in <class> class**

Error

The OS conformance class is defined as one of the BCC classes, but extended task is defined in the application via EVENT object.

TD0056: **ISR category 1 cannot have RESOURCE reference**

Error

The RESOURCE reference cannot be defined for ISR object if its *CATEGORY* attribute is 1.

TD0057: **ISR category 1 cannot use MESSAGE, ACCESSOR attribute shall be undefined**

Error

The message is generated if the *CATEGORY* attribute of ISR object is set to 1and ACCESSOR attribute is defined.

TD0058: **ISR cannot have reference to INTERNAL RESOURCE**

Error

The message is generated if the *RESOURCE* attribute of the ISR object refers to the *RESOURCE* object with the *RESOURCEPROPERTY* attribute with INTERNAL value.

TD0059: ISR cannot send/receive QUEUED MESSAGE

Error

The *MESSAGE* reference of the ISR object cannot refer to the MESSAGE object with the QUEUED value of the *TYPE* attribute.

TD0060: Accessor for QUEUED MESSAGE must be with copy only

Error

If the *MESSAGE* reference of the TASK object refers to the MESSAGE object with the QUEUED value of the *TYPE* attribute, then the *WITHOUTCOPY* attribute of the TASK object cannot be TRUE.

TD0065: Basic TASK cannot be notified by SETEVENT method

Error

If the referenced task has no events, then *ACTION* attribute of ALARM and MESSAGE objects cannot be defined as SETEVENT.

TD0066: TASK has no EVENT reference

Information

The referenced task is a basic one.

TD0070: TASK to be notified shall be defined

Error

The TASK reference is not defined.

TD0071: EVENT shall be defined for SETEVENT notification

Error

The EVENT reference shall be specified only if the *ACTION* attribute of the ALARM and MESSAGE objects is set as SETEVENT.

TD0072: EVENT <name> does not belong to TASK <name>

Warning

The task has no event, which is referenced by the EVENT reference of the ALARM object.

TD0073: More than one TASK per priority is defined for <class>

Warning

Only one task per priority may be used in BCC1 and ECC1 classes, i.e., each task should have unique priority.

TD0085: **Static stack size must be defined for Extended task**

Error

The task is considered as extended, if any event is referenced. For extended tasks the stack size should be defined. The *STACKSIZE* attribute of the *TASK* object should be defined.

TD0086: **Basic task doesn't require stack size**

Warning

The value of the *STACKSIZE* attribute is ignored for basic tasks.

TD0098: **ACTION = SETEVENT cannot be defined in <class> class**

Error

The event setting task notification method cannot be used in Basic conformance classes.

TD0101: **Name or ACCESSNAME shall be C-identifier**

Error

Only C-identifier can be used in the *Name* or *ACCESSNAME* attributes' names.

TD0107: **Identifier is longer than 32. It can cause compilation problem**

Warning

If object identifier is longer than 32 characters, the result of compilation depends of compiler used. Make identifier shorter if you got error during compilation of configuration or application source files.

TD0109: **MAXALLOWEDVALUE of hardware counter is assumed to be <number>. It will be changed**

Warning

Hardware counter has hardware defined maximal value for counter, so it cannot differ from assumed value. Counter information available via API will be changed to assumed value.

TD0110: **MESSAGE is declared but never used**

Warning

This message is generated when the MESSAGE object is declared but TASK or ISR objects don't declare accessors to this MESSAGE.

TD0111: **Accessor name <name> is not unique**

Error

The *ACCESSNAME* attributes should have the different values.

TD0112: **Step ordering number <number> is not unique**

Error

Two or more StepNumbers share the same number. The *StepNumber* attributes shall have the unique values.

TD0113: **One action has to be specified**

Error

Only one *ACTION* attribute has to be defined for the *MESSAGE* object in CCCB class.

TD0115: **Sum of step times is not equal to ScalePeriod**

Warning

The value of the *ScalePeriod* attribute does not correspond to sum of values of *StepTime* attributes.

TD0116: **One OS/APPMODE shall be defined**

Error

One object of OS and at least one APPMODE types shall be defined in an application.

TD0117: TimeScale works only when SysTimer is HWCOUNTER

Error

The *TimeScale* attribute and its subattributes can be defined only if the *SysTimer* attribute has HWCOUNTER value.

TD0118: TimeScale cannot work on COUNTER which is referenced by ALARM

Error

Since the system timer and alarm cannot use the same counter in case of TimeScale mechanism using, the *SysTimer* attribute of OS objects and *ALARM* objects cannot have the COUNTER attribute with the same value.

TD0120: **ResourceScheduler is FALSE but RES_SCHEDULER is defined. Definition ignored**

Warning

RES_SCHEDULER is supported if the *ResourceScheduler* attribute is TRUE.

TD0123: TASK can have one INTERNAL RESOURCE

Error

The *TASK* object can have only one reference to the *RESOURCE* object which has the *RESOURCEPROPERTY* subattribute with the INTERNAL value.

TD0125: **More than one ACCESSOR = SENT is defined for message**

Error

The message is generated if there is more than one *TASK* or *ISR* defines an accessor to write the same message. By other words, every message might be referred only by one *MESSAGE* subattribute within *ACCESSOR* = SENT defined for *TASK* or *ISR* object.

TD0126: **An ACCESSOR = SENT shall be defined for message**

Warning

The message is generated if there are neither *TASK* nor *ISR* defines an accessor to write the message. By other words, every message must be referred by one *MESSAGE* subattribute within the *ACCESSOR* = SENT defined for *TASK* or *ISR* object.

TD0134: **At least one StepTime shall be bigger than 0**

Error

The message is generated if all *Step*s in the *TimeScale* definition has the *StepTime* values equal to 0.

TD0200: **This Target-Specific DLL is not intended to work with this platform**

Fatal Error

The target-specific DLL does not correspond to target platform.

TD0201: <Target Name> target is not supported by your license!

Fatal Error

The license for target platform is not provided.

TD0202: **License <OSEKTURBO> expires in <number> days**
Warning

The message warns when the license expires in less than 30 days. The warning appears every work session.

TD0203: **License <OSEKTURBO> will expire tonight at midnight**
Warning

This warning appears when the license expires at this day.

TD0250: Attribute <name> is not converted
Warning

The message is generated if the attribute defined for implementation according to OIL 2.0/2.1 standard, cannot be converted to an attribute applicable to OIL 2.2 implementation. This attribute does not exist in the new implementation, and source and target implementations have not equivalent attribute.

TD0251: Cannot set to value <value>
Warning

The message is generated if the attribute defined for implementation according to OIL 2.0/2.1 standard, being converted to an attribute applicable to OIL 2.2 implementation, cannot accept old value. It can be caused out of range for new attribute.

TD0500: **ORTIFULL license not found. ORTI file will not be generated**
Warning

ORTIFULL license shall be installed for ORTI files generation.

TD0501: **EVENT masks don't arrange into 32 bits**
Error

The number of supported event is 32 per task, but this error can be generated also in case if some events have AUTO mask but other events have user defined mask with several bits set.

TD0507: RESOURCE is declared but never used

Error

The resource Ceiling priority is calculated automatically on the basis of information about priorities of tasks using the resource. If RESOURCE object is defines but not used, calculation of resource Ceiling priority is incorrect.

TD0509: RESOURCE with property INTERNAL can not be linked

Error

This message is generated if a *RESOURCE* object with *RESOURCEPROPERTY* = **LINKED** refers through the *LINKEDRESOURCE* subattribute to a *RESOURCE* object with *RESOURCEPROPERTY* = **INTERNAL**.

TD0513: <value> must be defined as SysTimer

Error

The *SecondTimer* attribute can be SWCOUNTER if the *SysTimer* is equal to SWCOUNTER or HWCOUNTER. The *SecondTimer* can be HWCOUNTER if the *SysTimer* attribute is HWCOUNTER.

TD0515: SecondTimer hardware cannot be the same as SysTimer

Error

The *TimerHardware* attributes inside *SysTimer* and *SecondTimer* attributes cannot have the same value.

TD0516: Prescaler attributes shall be both USER or OS

Error

The *Prescaler* attribute in the *SysTimer* attribute scope and the Prescaler attribute inside the *SecondTimer* attribute shall have the same value.

TD0517: Period shall be defined because of Prescaler.Value is AUTO

Error

The *Period* attribute shall be defined if the *Value* or/and *TimerModuloValue* attributes inside *SysTimer* attribute is AUTO.

TD0518: **Period shall be defined because of TimerModuloValue is AUTO**

Error

The *Period* attribute shall be defined if the *Value* or/and *TimerModuloValue* attributes inside *SysTimer* attribute is AUTO.

TD0519: **Cannot calculate with defined TimerModuloValue**

Error

If prescaler value is AUTO and timer modulo value is defined by the user, then prescaler cannot be calculated during system generation.

TD0520: **Prescaler.Values shall be identical for both timers**

Error

The *Prescaler* attribute in the *SysTimer* attribute scope and the *Prescaler* attribute inside the *SecondTimer* attribute shall have the same value.

TD0521: **Modulo value cannot fit in range. Period is too long**

Error

The message is generated if the modulo value is more than hardware allows for any prescaler value.

TD0522: **Modulo value cannot fit in range. Period is too short**

Error

The message is generated if the modulo value is less than hardware allows for any prescaler value.

TD0523: **StepTime value is too big**

Error

Specified value of the *StepTime* attribute cannot be initialized in hardware.

TD0524: **StepTime value cannot be represented precisely in hardware settings**

Error

The value of *StepTime* attribute differs from time that can be represented in hardware.

TD0525: **Specify StepTime value in multiples of <number> nanoseconds**

Information

The value of the *StepTime* attribute should be a multiple of number in nanoseconds, that is specified by the value of the *Period* attribute defined for the *SysTimer* attribute with HWCOUNTER value.

TD0526: **StepTime value exceeds maximal possible hardware counter period**

Error

The message is generated when calculated StepTime attribute's value is greater than COUNTER/MAXALLOWEDVALUE value which is referenced with OS/SysTimer/COUNTER.

TD0527: **Hardware counter period shall be at least <number> times greater**

Information

The message is generated after group of TD0526 error messages and specifies required increase of SysTimer/Period or Prescaler/Value attribute's value to eliminate TD0526 errors.

TD0740: **Cannot find registry key**

Fatal Error

The Windows Registry is inconsistent.

TD0741: **Cannot find registry value**

Fatal Error

The Windows Registry is inconsistent.

TD0750: **EVENT <name> and EVENT <name> share the same bit and used by TASK all together**

Warning

Sharing the same bit by event masks inside the task can cause that the task will obtain the unexpected event notification.

TD0752: **EVENT is declared but never used**

Warning

The EVENT object is defined in OIL file, but other OIL objects (task, alarms) have no references to this event.

TD0753: **Period is out of representable range**
Warning

The message is generated if the period size is more than 32 bits.

TD0754: **Period is defined by prescaler and timer modulo. It will be ignored**
Warning

The *Period* attribute is ignored if *Prescaler/Value* and *TimerModuloValue* attributes don't have AUTO value.

TD0755: **The name of OSEK OS property file is not defined explicitly**
Warning

The **-p** option is used to define location of OS property file.

TD0756: **The name of OSEK OS header configuration file is not defined explicitly**
Warning

The **-h** option is used to define the name of OS header file.

TD0757: **The name of OSEK OS source configuration file is not defined explicitly**
Warning

The **-c** option is used to define the name of OS source file.

TD0758: **The name of OSEK ORTI file is not defined explicitly**
Warning

The **-o** option is used to define the name of ORTI file.

TD0759: **DEBUG_LEVEL equals 0. Option -o is ignored. ORTI file will not be generated**
Warning

If the *DEBUG_LEVEL* attribute is set 0, then ORTI is not supported, and ORTI file is not generated.

TD0760: **<filename> shall be used for OSEK OS property file name**
Information

If the location of osprop.h file is not defined explicitly, then by default this file is located in the source OIL file directory.

The **-p** option is used to define the location of OS property file.

TD0761: **<filename> shall be used for OSEK OS header configuration file name**

Information

If the name of header file is not defined explicitly, then by default this file has the same name as the source OIL file, .h extension and located in the same directory.

The **-h** option is used to define the name of OS header file.

TD0762: **<filename> shall be used for OSEK OS source configuration file name**

Information

If the name of source file is not defined explicitly, then by default this file has the same name as the source OIL file, .c extension and located in the same directory.

The **-c** option is used to define the name of OS source file.

TD0763: **<filename> shall be used for OSEK ORTI file name**

Information

If the name of ORTI file is not defined explicitly, then by default this file has the same name as the source OIL file, .ort extension and located in the same directory.

The **-o** option is used to define the name of ORTI file.

TD0765: **EVENT is treated as it is used by all the TASKs**

Information

The EVENT object is defined in OIL file, but task objects have no references to this event. This leads for allocation of the bit in event mask which cannot be used by other event masks.

TD0766: EVENT is declared but never used and there are no Extended TASKs

Error

The EVENT object is defined in OIL file, but task have no references to this event. The task is considered as extended, if any event is referenced.

TD0767: **The version of ORTI specification is not defined explicitly, version 2.1 will be used**

Warning

If the version of ORTI file format is not defined explicitly, then by default this file will be generated in format corresponding to version 2.1.

The **-O** option is used to define the version of ORTI file format.

TD0768: **-O option requires either 2.0 or 2.1 version of ORTI specification as a parameter**

Error

The current OSEKturbo version supports ORTI specification version 2.0 and 2.1.

TD0775: **Specified Period cannot be represented in hardware. Period <period value> is calculated**

Warning

If the *Period* specified by user cannot be represented precisely in hardware, Target DLL calculates closest period to it.

TD0800: **Number of priorities exceeds <number>**

Error

For OS/MPC number of priorities is limited by 64.

TD0801: Number of RESOURCEs exceeds <number>

Error

Number of RESOURCE objects defined in system is restricted by 254 (including RES_SCHEDULER).

TD0802: **Number of ALARMs exceeds 254**

Error

Number of ALARM objects defined in system is restricted by 254.

TD0803: **Number of MESSAGEs exceeds 254**

Error

Number of MESSAGE objects defined in system is restricted by 254.

TD0804: **Number of COUNTERs exceeds 254**

Error

Number of COUNTER objects defined in system is restricted by 254.

TD805: **ALARMTIME for autostarted alarm can not be more than MAXALLOWEDVALUE of COUNTER**

Error

The value of the AUTOSTART/ALARMTIME attribute of the ALARM object shall be less than the value of the MAXALLOWEDVALUE attribute of the COUNTER object.

TD806: **CYCLETIME for autostarted alarm shall be more than MINCYCLE and less than MAXALLOWEDVALUE of COUNTER**

Error

The value of the AUTOSTART/CYCLETIME attribute of the ALARM object shall be more than the value of the MINCYCLE attribute and less than the value of the MAXALLOWEDVALUE attribute of the COUNTER object.

TD0807: Number of APPMODEs exceeds 8

Error

Number of APPMODE objects defined in system is restricted by 8.

TD0808: APPMODE attribute shall be defined

Error

The APPMODE attribute shall be defined for TASK/ALARM with AUTOSTART = TRUE if there are more than one APPMODE defined in application.

TD1500: **Prescaler value of <name> timer for <name> MCU cannot be more than <number>**

Error

MCU manufacturer reserves values of the prescaler.

TD1501: **Prescaler value 0 means RTI timer is OFF**

Error

If the RTI timer is not used, then you should disable this timer.

TD2000: **There are no defined ISRs. InterruptDispatcher is switched from <value> to None**

Warning

If the no ISR objects and no timers are defined in the application definition, then *InterruptDispatcher* attribute's value will be changed to None.

TD2001: **Only even ISR priorities must be used for timers [0..14]**

Error

PIT and RTC timers for MPC555 can use only even priorities between [0..14] range.

TD2002: **ISR priority for DEC timer must be equal to <value>**

Error

DEC timer for MPC555 must have ISR priority is equal to 16.

TD2003: **Only one ISR per priority is allowed for InterruptDispatcher =<value>**

Error

If the InterruptDispatcher attribute has MultiLevel or OneLevel value, then each ISR object must have unique priority.

TD2004: **Values of RTSEL for PIT and RTC (RTC and PIT) timers must be equivalent**

Error

They must be either OSCM or EXTCLK both because they represent one hardware bit.

TD2006: **ISR priorities of Time Base counters must be equal**

Error

The ISRPRIORITY attributes shall have the equal values because they concern to same timer ISR.

TD2251: InterruptDispatcher can be defined only for derivative

Error

The *InterruptDispatcher* attribute has to be defined as None if the TargetMCU value corresponds CPU family name - MPC.

TD2252: Enhanced InterruptDispatcher can be defined only for MPC565 or MPC561

Error

The Enchanced value is available for the InterruptDispatcher attribute only if the TargetMCU attribute has MPC561, MPC563 or MPC565 value.

Index